Tell us about your PDF experience.

Azure Cognitive Search documentation

Cloud search over private heterogeneous content, with options for AI enrichment if your content is unstructured or unsearchable in raw form.

About Cognitive Search

OVERVIEW

What is Azure Cognitive Search?

What is semantic search?

TRAINING

Learning module

10-minute quickstarts

QUICKSTART

Create a search service

Create a search index

Create a demo app

Create a skillset

Develop apps

TUTORIAL

Add search to web apps

HOW-TO GUIDE

Develop in .NET

Develop in REST

REFERENCE

REFERENCE

Azure REST API Reference

Azure SDK for .NET

Azure SDK for Python

Index data

OVERVIEW

What's a search index?

Importing data

Indexer overview

HOW-TO GUIDE

Index from Azure Blob Storage

Index from Azure SQL Database

Index from Azure Cosmos DB

Index any data

Query data

CONCEPT

Query types and composition

Simple syntax (default)

Full Lucene syntax

HOW-TO GUIDE

Create a simple query

Create advanced queries

REFERENCE

OData language reference

Search Documents (REST)

Enrich with AI

CONCEPT

What is AI enrichment?

Skillsets

Knowledge stores

What's Azure Cognitive Search?

Article • 12/02/2022 • 7 minutes to read

Azure Cognitive Search (formerly known as "Azure Search") is a cloud search service that gives developers infrastructure, APIs, and tools for building a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

Search is foundational to any app that surfaces text to users, where common scenarios include catalog or document search, online retail apps, or data exploration over proprietary content. When you create a search service, you'll work with the following capabilities:

- A search engine for full text search over a search index containing user-owned content
- Rich indexing, with lexical analysis and optional AI enrichment for content extraction and transformation
- Rich query syntax for text search, fuzzy search, autocomplete, geo-search and more
- Programmability through REST APIs and client libraries in Azure SDKs
- Azure integration at the data layer, machine learning layer, and AI (Cognitive Services)

Create a search service

Architecturally, a search service sits between the external data stores that contain your un-indexed data, and your client app that sends query requests to a search index and handles the response.

In your client app, the search experience is defined using APIs from Azure Cognitive Search, and can include relevance tuning, semantic ranking, autocomplete, synonym matching, fuzzy matching, pattern matching, filter, and sort.

Across the Azure platform, Cognitive Search can integrate with other Azure services in the form of *indexers* that automate data ingestion/retrieval from Azure data sources, and *skillsets* that incorporate consumable AI from Cognitive Services, such as image and

natural language processing, or custom AI that you create in Azure Machine Learning or wrap inside Azure Functions.

Inside a search service

On the search service itself, the two primary workloads are *indexing* and *querying*.

- **Indexing** is an intake process that loads content into your search service and makes it searchable. Internally, inbound text is processed into tokens and stored in inverted indexes for fast scans. You can upload JSON documents, or use an indexer to serialize your data into JSON.

  AI enrichment through cognitive skills is an extension of indexing. If your content needs image or language analysis before it can be indexed, AI enrichment can extract text embedded in application files, translate text, and also infer text and structure from non-text files by analyzing the content.

- **Querying** can happen once an index is populated with searchable text, when your client app sends query requests to a search service and handles responses. All query execution is over a search index that you control.

  Semantic search is an extension of query execution. It adds language understanding to search results processing, promoting the most semantically relevant results to the top.

Why use Cognitive Search?

Azure Cognitive Search is well suited for the following application scenarios:

- Consolidate heterogeneous content into a private, user-defined search index.
- Offload indexing and query workloads onto a dedicated search service.
- Easily implement search-related features: relevance tuning, faceted navigation, filters (including geo-spatial search), synonym mapping, and autocomplete.
- Transform large undifferentiated text or image files, or application files stored in Azure Blob Storage or Azure Cosmos DB, into searchable chunks. This is achieved during indexing through cognitive skills that add external processing.
- Add linguistic or custom text analysis. If you have non-English content, Azure Cognitive Search supports both Lucene analyzers and Microsoft's natural language processors. You can also configure analyzers to achieve specialized processing of

raw content, such as filtering out diacritics, or recognizing and preserving patterns in strings.

For more information about specific functionality, see Features of Azure Cognitive Search

How to get started

Functionality is exposed through the Azure portal, simple REST APIs, or Azure SDKs like the Azure SDK for .NET. The Azure portal supports service administration and content management, with tools for prototyping and querying your indexes and skillsets.

An end-to-end exploration of core search features can be accomplished in four steps:

1. **Decide on a tier** and region. One free search service is allowed per subscription. All quickstarts can be completed on the free tier. For more capacity and capabilities, you'll need a billable tier.

2. **Create a search service** in the Azure portal.

3. **Start with Import data wizard**. Choose a built-in sample or a supported data source to create, load, and query an index in minutes.

4. **Finish with Search Explorer**, using a portal client to query the search index you just created.

Alternatively, you can create, load, and query a search index in atomic steps:

1. **Create a search index** using the portal, REST API, .NET SDK, or another SDK. The index schema defines the structure of searchable content.

2. **Upload content** using the "push" model to push JSON documents from any source, or use the "pull" model (indexers) if your source data is of a supported type.

3. **Query an index** using Search explorer in the portal, REST API, .NET SDK, or another SDK.

> **Tip**
>
> For help with complex or custom solutions, **contact a partner** with deep expertise in Cognitive Search technology.

Compare search options

Customers often ask how Azure Cognitive Search compares with other search-related solutions. The following table summarizes key differences.

Compared to	Key differences
Microsoft Search	Microsoft Search is for Microsoft 365 authenticated users who need to query over content in SharePoint. It's offered as a ready-to-use search experience, enabled and configured by administrators, with the ability to accept external content through connectors from Microsoft and other sources. If this describes your scenario, then Microsoft Search with Microsoft 365 is an attractive option to explore. In contrast, Azure Cognitive Search executes queries over an index that you define, populated with data and documents you own, often from diverse sources. Azure Cognitive Search has crawler capabilities for some Azure data sources through indexers, but you can push any JSON document that conforms to your index schema into a single, consolidated searchable resource. You can also customize the indexing pipeline to include machine learning and lexical analyzers. Because Cognitive Search is built to be a plug-in component in larger solutions, you can integrate search into almost any app, on any platform.
Bing	Bing Web Search API searches the indexes on Bing.com for matching terms you submit. Indexes are built from HTML, XML, and other web content on public sites. Built on the same foundation, Bing Custom Search offers the same crawler technology for web content types, scoped to individual web sites. In Cognitive Search, you can define and populate the index. You can use indexers to crawl data on Azure data sources, or push any index-conforming JSON document to your search service.

Compared to	Key differences
Database search	Many database platforms include a built-in search experience. SQL Server has full text search. Azure Cosmos DB and similar technologies have queryable indexes. When evaluating products that combine search and storage, it can be challenging to determine which way to go. Many solutions use both: DBMS for storage, and Azure Cognitive Search for specialized search features. Compared to DBMS search, Azure Cognitive Search stores content from heterogeneous sources and offers specialized text processing features such as linguistic-aware text processing (stemming, lemmatization, word forms) in 56 languages. It also supports autocorrection of misspelled words, synonyms, suggestions, scoring controls, facets, and custom tokenization. The full text search engine in Azure Cognitive Search is built on Apache Lucene, an industry standard in information retrieval. However, while Azure Cognitive Search persists data in the form of an inverted index, it isn't a replacement for true data storage and we don't recommend using it in that capacity. For more information, see this forum post. Resource utilization is another inflection point in this category. Indexing and some query operations are often computationally intensive. Offloading search from the DBMS to a dedicated solution in the cloud preserves system resources for transaction processing. Furthermore, by externalizing search, you can easily adjust scale to match query volume.
Dedicated search solution	Assuming you've decided on dedicated search with full spectrum functionality, a final categorical comparison is between on premises solutions or a cloud service. Many search technologies offer controls over indexing and query pipelines, access to richer query and filtering syntax, control over rank and relevance, and features for self-directed and intelligent search. A cloud service is the right choice if you want a turn-key solution with minimal overhead and maintenance, and adjustable scale. Within the cloud paradigm, several providers offer comparable baseline features, with full-text search, geospatial search, and the ability to handle a certain level of ambiguity in search inputs. Typically, it's a specialized feature, or the ease and overall simplicity of APIs, tools, and management that determines the best fit.

Among cloud providers, Azure Cognitive Search is strongest for full text search workloads over content stores and databases on Azure, for apps that rely primarily on search for both information retrieval and content navigation.

Key strengths include:

- Data integration (crawlers) at the indexing layer.
- AI and machine learning integration with Azure Cognitive Services, useful if you need to make unsearchable content full text-searchable.

- Security integration with Azure Active Directory for trusted connections, and with Azure Private Link integration to support private connections to a search index in no-internet scenarios.
- Linguistic and custom text analysis in 56 languages.
- Full search experience: rich query language, relevance tuning and semantic ranking, faceting, autocomplete queries and suggested results, and synonyms.
- Azure scale, reliability, and world-class availability.

Among our customers, those able to apply the widest range of features in Azure Cognitive Search include online catalogs, line-of-business programs, and document discovery applications.

What's new in Azure Cognitive Search

Article • 05/15/2023

Learn about the latest updates to Azure Cognitive Search functionality, docs, and samples.

> **Note**
>
> Looking for preview features? Previews are announced here, but we also maintain a **preview features list** so you can find them in one place.

May 2023

Item	Type	Description
Azure RBAC (role-based access control)	Feature	Announcing general availability.
2022-09-01 Management REST API	API	New stable version of the Management REST APIs, with support for configuring search to use Azure RBAC. The **Az.Search** module of Azure PowerShell and **Az search** module of the Azure CLI are updated to support search service authentication options. You can also use the **Terraform provider** to configure authentication options (see this Terraform quickstart for details).

April 2023

Item	Type	Description
Multi-region deployment of Azure Cognitive Search for business continuity and disaster recovery	Sample	Deployment scripts that fully configure a multi-regional solution for Azure Cognitive Search, with options for synchronizing content and request redirection if an endpoint fails.

March 2023

Item	Type	Description

Item	Type	Description
ChatGPT + Enterprise data with Azure OpenAI and Cognitive Search (GitHub)	Sample	Python code and a template for combining Cognitive Search with the large language models in OpenAI. For background, see this Tech Community blog post: Revolutionize your Enterprise Data with ChatGPT. Key points: Use Cognitive Search to consolidate and index searchable content. Query the index for initial search results. Assemble prompts from those results and send to the gpt-35-turbo (preview) model in Azure OpenAI. Return a cross-document answer and provide citations and transparency in your customer-facing app so that users can assess the response.

2022 announcements

Month	Item
November	**Add search to websites** series, updated versions of React and Azure SDK client libraries: • C# • Python • JavaScript "Add search to websites" is a tutorial series with sample code available in three languages. If you're integrating client code with a search index, these samples demonstrate an end-to-end approach to integration.
November	**Retired** - Visual Studio Code extension for Azure Cognitive Search.
November	Query performance dashboard. This Application Insights sample demonstrates an approach for deep monitoring of query usage and performance of an Azure Cognitive Search index. It includes a JSON template that creates a workbook and dashboard in Application Insights and a Jupyter Notebook that populates the dashboard with simulated data.
October	Compliance risk analysis using Azure Cognitive Search. Published on Azure Architecture Center, this guide covers the implementation of a compliance risk analysis solution that uses Azure Cognitive Search.

Month	Item
October	Beiersdorf customer story using Azure Cognitive Search. This customer story showcases semantic search and document summarization to provide researchers with ready access to institutional knowledge.
September	Azure Cognitive Search Lab. This C# sample provides the source code for building a web front-end that accesses all of the REST API calls against an index. This tool is used by support engineers to investigate customer support issues. You can try this demo site before building your own copy.
September	Event-driven indexing for Cognitive Search. This C# sample is an Azure Function app that demonstrates event-driven indexing in Azure Cognitive Search. If you've used indexers and skillsets before, you know that indexers can run on demand or on a schedule, but not in response to events. This demo shows you how to set up an indexing pipeline that responds to data update events.
August	Tutorial: Index large data from Apache Spark. This tutorial explains how to use the SynapseML open-source library to push data from Apache Spark into a search index. It also shows you how to make calls to Cognitive Services to get AI enrichment without skillsets and indexers.
June	Semantic search (preview). New support for Storage Optimized tiers (L1, L2).
June	**General availability** - Debug Sessions.
May	**Retired** - Power Query connector preview.
February	Index aliases. An index alias is a secondary name that can be used to refer to an index for querying, indexing, and other operations. When index names change, for example if you version the index, instead of updating the references to an index name in your application, you can just update the mapping for your alias.

Previous year's announcements

- 2021 announcements
- 2020 announcements
- 2019 announcements

Service re-brand

Azure Search was renamed to **Azure Cognitive Search** in October 2019 to reflect the expanded (yet optional) use of cognitive skills and AI processing in service operations.

Service updates

Service update announcements for Azure Cognitive Search can be found on the Azure web site.

Features of Azure Cognitive Search

Article • 05/05/2023

Azure Cognitive Search provides a full-text search engine, persistent storage of search indexes, integrated AI used during indexing to extract more text and structure, and APIs and tools.

The following table summarizes features by category. For more information about how Cognitive Search compares with other search technologies, see Compare search options.

Indexing features

Category	Features
Data sources	Search indexes can accept text from any source, provided it's submitted as a JSON document. **Indexers** are a feature that automates data import from supported data sources to extract searchable content in primary data stores. Indexers handle JSON serialization for you and most support some form of change and deletion detection. You can connect to a variety of data sources, including Azure SQL Database, Azure Cosmos DB, or Azure Blob storage.
Hierarchical and nested data structures	**Complex types** and collections allow you to model virtually any type of JSON structure within a search index. One-to-many and many-to-many cardinality can be expressed natively through collections, complex types, and collections of complex types.
Linguistic analysis	Analyzers are components used for text processing during indexing and search operations. By default, you can use the general-purpose Standard Lucene analyzer, or override the default with a language analyzer, a custom analyzer that you configure, or another predefined analyzer that produces tokens in the format you require. **Language analyzers** from Lucene or Microsoft are used to intelligently handle language-specific linguistics including verb tenses, gender, irregular plural nouns (for example, 'mouse' vs. 'mice'), word de-compounding, word-breaking (for languages with no spaces), and more. **Custom lexical analyzers** are used for complex query forms such as phonetic matching and regular expressions.

AI enrichment and knowledge mining

Category	Features
AI processing during indexing	**AI enrichment** refers to embedded image and natural language processing in an indexer pipeline that extracts text and information from content that can't otherwise be indexed for full text search. AI processing is achieved by adding and combining skills in a skillset, which is then attached to an indexer. AI can be either built-in skills from Microsoft, such as text translation or Optical Character Recognition (OCR), or custom skills that you provide.
Storing enriched content for analysis and consumption in non-search scenarios	**Knowledge store** is persistent storage of enriched content, intended for non-search scenarios like knowledge mining and data science processing. A knowledge store is defined in a skillset, but created in Azure Storage as objects or tabular rowsets.
Cached enrichments	**Incremental enrichment (preview)** refers to cached enrichments that can be reused during skillset execution. Caching is particularly valuable in skillsets that include OCR and image analysis, which are expensive to process.

Query and user experience

Category	Features
Free-form text search	**Full-text search** is a primary use case for most search-based apps. Queries can be formulated using a supported syntax. **Simple query syntax** provides logical operators, phrase search operators, suffix operators, precedence operators. **Full Lucene query syntax** includes all operations in simple syntax, with extensions for fuzzy search, proximity search, term boosting, and regular expressions.

Category	Features
Relevance	**Simple scoring** is a key benefit of Azure Cognitive Search. Scoring profiles are used to model relevance as a function of values in the documents themselves. For example, you might want newer products or discounted products to appear higher in the search results. You can also build scoring profiles using tags for personalized scoring based on customer search preferences you've tracked and stored separately. **Semantic search (preview)** is premium feature that reranks results based on semantic relevance to the query. Depending on your content and scenario, it can significantly improve search relevance with almost minimal configuration or effort.
Geospatial search	**Geospatial functions** filter over and match on geographic coordinates. You can match on distance or by inclusion in a polygon shape.
Filters and facets	**Faceted navigation** is enabled through a single query parameter. Azure Cognitive Search returns a faceted navigation structure you can use as the code behind a categories list, for self-directed filtering (for example, to filter catalog items by price-range or brand). **Filters** can be used to incorporate faceted navigation into your application's UI, enhance query formulation, and filter based on user- or developer-specified criteria. Create filters using the OData syntax.

Category	Features
User experience	Autocomplete can be enabled for type-ahead queries in a search bar. Search suggestions also works off of partial text inputs in a search bar, but the results are actual documents in your index rather than query terms. Synonyms associates equivalent terms that implicitly expand the scope of a query, without the user having to provide the alternate terms. Hit highlighting applies text formatting to a matching keyword in search results. You can choose which fields return highlighted snippets. Sorting is offered for multiple fields via the index schema and then toggled at query-time with a single search parameter. Paging and throttling your search results is straightforward with the finely tuned control that Azure Cognitive Search offers over your search results.

Security features

Category	Features
Data encryption	Microsoft-managed encryption-at-rest is built into the internal storage layer and is irrevocable. Customer-managed encryption keys that you create and manage in Azure Key Vault can be used for supplemental encryption of indexes and synonym maps. For services created after August 1 2020, CMK encryption extends to data on temporary disks, for full double encryption of indexed content.
Endpoint protection	IP rules for inbound firewall support allows you to set up IP ranges over which the search service will accept requests. Create a private endpoint using Azure Private Link to force all requests through a virtual network.

Category	Features
Inbound access	Azure role-based access control assigns roles to users and groups in Azure Active Directory for controlled access to search content and operations. You can also use key-based authentication if you don't have an Azure tenant.
Outbound security (indexers)	Data access through private endpoints allows an indexer to connect to Azure resources that are protected through Azure Private Link. Data access using a trusted identity means that connection strings to external data sources can omit user names and passwords. When an indexer connects to the data source, the resource allows the connection if the search service was previously registered as a trusted service.

Portal features

Category	Features
Tools for prototyping and inspection	Add index is an index designer in the portal that you can use to create a basic schema consisting of attributed fields and a few other settings. After saving the index, you can populate it using an SDK or the REST API to provide the data. Import data wizard creates indexes, indexers, skillsets, and data source definitions. If your data exists in Azure, this wizard can save you significant time and effort, especially for proof-of-concept investigation and exploration. Search explorer is used to test queries and refine scoring profiles. Create demo app is used to generate an HTML page that can be used to test the search experience. Debug Sessions is a visual editor that lets you debug a skillset interactively. It shows you dependencies, output, and transformations.
Monitoring and diagnostics	Enable monitoring features to go beyond the metrics-at-a-glance that are always visible in the portal. Metrics on queries per second, latency, and throttling are captured and reported in portal pages with no extra configuration required.

Programmability

Category	Features
REST	Service REST API is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics. Management REST API is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.
Azure SDK for .NET	Azure.Search.Documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics. Microsoft.Azure.Management.Search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.
Azure SDK for Java	com.azure.search.documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics. com.microsoft.azure.management.search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.
Azure SDK for Python	azure-search-documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics. azure-mgmt-search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.
Azure SDK for JavaScript/TypeScript	azure/search-documents is for data plane operations, including all operations related to indexing, queries, and AI enrichment. You can also use this client library to retrieve system information and statistics. azure/arm-search is for service creation and provisioning through Azure Resource Manager. You can also use this API to manage keys and capacity.

See also

- What's new in Cognitive Search
- Preview features in Cognitive Search

Cognitive Search Frequently Asked Questions

FAQ

Find answers to commonly asked questions about Azure Cognitive Search.

General

What is Azure Cognitive Search?

Azure Cognitive Search is a service on Azure that provides a dedicated search engine and persistent storage of your searchable content for full text search scenarios. It also includes optional, integrated AI used during indexing to extract more text and structure from raw content.

How do I work with Cognitive Search?

The primary workflow is create, load, and query an index. Although you can use the portal for most tasks, Cognitive Search is intended to be used programmatically, handling requests from client code. Programmatic support is provided through REST APIs and client libraries in .NET, Python, Java, and JavaScript SDKs for Azure.

Are "Azure Search" and "Azure Cognitive Search" the same service?

Azure Search was renamed to Azure Cognitive Search in October 2019 to reflect the expanded (yet optional) use of cognitive skills and AI processing in service operations.

What languages are supported?

The default analyzer used for tokenization is standard Lucene and it's language agnostic. Otherwise, language support is expressed through language analyzers that apply linguistic rules to inbound (indexing) and outbound (queries) content. Some features, such as semantic search and speller, are limited to a subset of languages.

How do I integrate search into my solution?

Client code should call the client libraries or REST APIs to connect to a search index, formulate queries, and handle responses. You can also write code that builds and refreshes an index, or runs indexers programmatically or by script.

Is there functional parity across the various APIs?

Not always. The REST API is always the first to implement new features in preview API versions, and the generally available versions support all programmatic operations. The client libraries in Azure SDKs will pick up new features over time, but are released on their own schedule.

Although the REST APIs are first out with newest features, the Azure SDKs provide more coding support, and are recommended over REST unless a required feature is unavailable.

Can I pause the service and stop billing?

You can't pause a search service. In Azure Cognitive Search, computing resources are allocated when the service is created. It's not possible to release and reclaim those resources on-demand.

Can I upgrade, downgrade, rename or move the service?

Service tier, name, and region are fixed for the lifetime of the service.

If I migrate my search service to another subscription or resource group, should I expect any downtime?

As long as you follow the checklist before moving resources and make sure each step is completed, there shouldn't be any downtime.

Indexing

What does "indexing" mean in Cognitive Search?

It refers to the ingestion, parsing, and storing of textual content and tokens that populate a search index. Indexing creates inverted indices and other physical data

structures that support information retrieval.

Can I move, backup, and restore indexes?

There's no native support for index management. Search indexes are considered downstream data structures, accepting content from other data sources that collect operational data. As such, there's no built-in support for backing up and restoring indexes because the expectation is that you would rebuild an index from source data if you deleted it, or wanted to move it.

However, if you want to move an index between search services, you can try the **index-backup-restore** sample code in this Azure Cognitive Search .NET sample repo.

Can I restore my index or service once it's deleted?

No, if you delete an Azure Cognitive Search index or service, it can't be recovered. When you delete a search service, all indexes in the service are deleted permanently.

Can I index from SQL Database replicas?

If you're using the search indexer for Azure SQL Database, there are no restrictions on the use of primary or secondary replicas as a data source when building an index from scratch. However, refreshing an index with incremental updates (based on changed records) requires the primary replica. This requirement comes from SQL Database, which guarantees change tracking on primary replicas only. If you try using secondary replicas for an index refresh workload, there's no guarantee you get all of the data.

Queries

Where does query execution occur?

Queries execute over a single search index that's hosted on your search service. You can't join multiple indices to search content in two or more indexes, but you can query same-name indexes in multiple search services.

Why are there zero matches on terms I know to be valid?

The most common case isn't knowing that each query type supports different search behaviors and levels of linguistic analyses. Full text search, which is the predominant workload, includes a language analysis phase that breaks down terms to root forms. This aspect of query parsing casts a broader net over possible matches, because the tokenized term matches a greater number of variants.

Wildcard, fuzzy and regex queries, however, aren't analyzed like regular term or phrase queries and can lead to poor recall if the query doesn't match the analyzed form of the word in the search index. For more information on query parsing and analysis, see query architecture.

Why are my wildcard searches slow?

Most wildcard search queries, like prefix, fuzzy and regex, are rewritten internally with matching terms in the search index. This extra processing adds to latency. Further, broad search queries, like `a*` for example, that are likely to be rewritten with many terms can be slow. For performant wildcard searches, consider defining a custom analyzer.

Can I search across multiple indexes?

No, a query is always scoped to a single index.

Why is the search score a constant 1.0 for every match?

Search scores are generated for full text search queries, based on the statistical properties of matching terms, and ordered high to low in the result set. Query types that aren't full text search (wildcard, prefix, regex) aren't ranked by a relevance score. This behavior is by design. A constant score allow matches found through query expansion to be included in the results, without affecting the ranking.

For example, suppose an input of "tour*" in a wildcard search produces matches on "tours", "tourettes", and "tourmaline". Given the nature of these results, there's no way to reasonably infer which terms are more valuable than others. For this reason, term frequencies are ignored when scoring results in queries of types wildcard, prefix, and regex. Search results based on a partial input are given a constant score to avoid bias towards potentially unexpected matches.

Security

Where does Cognitive Search store customer data?

It stores your data wherever your service is deployed. Cognitive Search doesn't store customer data outside of the deployment region.

Does Cognitive Search send customer data to other services for processing?

Yes, if you use the built-in skills based on Cognitive Services, the indexer sends requests to Cognitive Services over the internal network. If you add a custom skill, the indexer will send content to the URI provided in the custom skill over the public network.

Can I control access to search results based on user identity?

Not exactly. Typically, users who are authorized to run your application are also authorized to see all search results. Cognitive Search doesn't have built-in support for row-level or document-level permissions, but you can implement security filters as a workaround.

Can I control access to operations based on user identity?

Yes, you can use role-based authorization for data plane operations over content.

Can I use the Azure portal to view and manage search content if the search service is behind an IP firewall or a private endpoint?

You can use the Azure portal on a network-protected search service if you create a network exception that allows client and portal access. For more information, see connect through an IP firewall or connect through a private endpoint.

Next steps

If your question isn't answered here, you can refer to the following sources for more questions and answers.

Stack Overflow: Azure Cognitive Search
How full text search works in Azure Cognitive Search
What is Azure Cognitive Search?

Quickstart: Create an Azure Cognitive Search index in the Azure portal

Article • 11/16/2022

In this quickstart, you'll create your first search index using the **Import data** wizard and a built-in sample data source consisting of fictitious hotel data. The wizard guides you through the creation of a search index (hotels-sample-index) so that you can write interesting queries within minutes.

Although you won't use the options in this quickstart, the wizard includes a page for AI enrichment so that you can extract text and structure from image files and unstructured text. For a similar walkthrough that includes AI enrichment, see Quickstart: Create a skillset.

Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An Azure Cognitive Search service (any tier, any region). Create a service or find an existing service under your current subscription. You can use a free service for this quickstart.

Check for space

Many customers start with the free service. The free tier is limited to three indexes, three data sources, and three indexers. Make sure you have room for extra items before you begin. This quickstart creates one of each object.

Check the service overview page to find out how many indexes, indexers, and data sources you already have.

Create and load an index

Search queries iterate over an *index* that contains searchable data, metadata, and other constructs that optimize certain search behaviors.

For this quickstart, we'll create and load the index using a built-in sample dataset that can be crawled using an *indexer* via the **Import data wizard**. The hotels-sample data set is hosted on Microsoft on Azure Cosmos DB and accessed over an internal connection. You don't need your own Cosmos DB account or source files to access the data.

An indexer is a source-specific crawler that can read metadata and content from supported Azure data sources. Normally, indexers are created programmatically, but in the portal, you can create them through the **Import data wizard**.

Step 1 - Start the Import data wizard and create a data source

1. Sign in to the Azure portal with your Azure account.

2. Find your search service and on the Overview page, select **Import data** on the command bar to create and populate a search index.

3. In the wizard, select **Connect to your data** > **Samples** > **hotels-sample**. This data source is built-in. If you were creating your own data source, you would need to specify a name, type, and connection information. Once created, it becomes an "existing data source" that can be reused in other import operations.

4. Continue to the next page.

Step 2 - Skip the "Enrich content" page

The wizard supports the creation of an AI enrichment pipeline for incorporating the Cognitive Services AI algorithms into indexing.

We'll skip this step for now, and move directly on to **Customize target index**.

> **Tip**
>
> You can step through an AI-indexing example in a **quickstart** or **tutorial**.

Step 3 - Configure index

For the built-in hotels sample index, a default index schema is defined for you. Except for a few advanced filter examples, queries in the documentation and samples that target the hotel-samples index will run on this index definition:

Typically, in a code-based exercise, index creation is completed prior to loading data. The Import data wizard condenses these steps by generating a basic index for any data source it can crawl. Minimally, an index requires a name and a fields collection. One of the fields should be marked as the document key to uniquely identify each document. Additionally, you can specify language analyzers or suggesters if you want autocomplete or suggested queries.

Fields have a data type and attributes. The check boxes across the top are *attributes* controlling how the field is used.

- **Key** is the unique document identifier. It's always a string, and it's required. Only one field can be the key.
- **Retrievable** means that field contents show up in search results list. You can mark individual fields as off limits for search results by clearing this checkbox, for example for fields used only in filter expressions.
- **Filterable**, **Sortable**, and **Facetable** determine whether fields are used in a filter, sort, or faceted navigation structure.
- **Searchable** means that a field is included in full text search. Strings are searchable. Numeric fields and Boolean fields are often marked as not searchable.

Storage requirements can vary as a result of attribute selection. For example, **Filterable** requires more storage, but **Retrievable** doesn't.

By default, the wizard scans the data source for unique identifiers as the basis for the key field. *Strings* are attributed as **Retrievable** and **Searchable**. *Integers* are attributed as **Retrievable**, **Filterable**, **Sortable**, and **Facetable**.

1. Accept the defaults.

If you rerun the wizard a second time using an existing hotels data source, the index won't be configured with default attributes. You'll have to manually select attributes on future imports.

2. Continue to the next page.

Step 4 - Configure indexer

Still in the **Import data** wizard, select **Indexer** > **Name**, and type a name for the indexer.

This object defines an executable process. You could put it on recurring schedule, but for now use the default option to run the indexer once, immediately.

Select **Submit** to create and simultaneously run the indexer.

Monitor progress

The wizard should take you to the Indexers list where you can monitor progress. For self-navigation, go to the Overview page and select the **Indexers** tab.

It can take a few minutes for the portal to update the page, but you should see the newly created indexer in the list, with status indicating "in progress" or success, along with the number of documents indexed.

Check results

The service overview page provides links to the resources created in your Azure Cognitive Search service. To view the index you just created, select **Indexes** from the list of links.

Wait for the portal page to refresh. After a few minutes, you should see the index with a document count and storage size.

From this list, you can select on the *hotels-sample* index that you just created, view the index schema. and optionally add new fields.

The **Fields** tab shows the index schema. If you're writing queries and need to check whether a field is filterable or sortable, this tab shows you the attributes.

Scroll to the bottom of the list to enter a new field. While you can always create a new field, in most cases, you can't change existing fields. Existing fields have a physical representation in your search service and are thus non-modifiable, not even in code. To fundamentally change an existing field, create a new index, dropping the original.

Other constructs, such as scoring profiles and CORS options, can be added at any time.

To clearly understand what you can and can't edit during index design, take a minute to view index definition options. Grayed-out options are an indicator that a value can't be modified or deleted.

Query using Search explorer

You now have a search index that can be queried using **Search explorer**.

Search explorer sends REST calls that conform to the Search Documents API. The tool supports simple query syntax and full Lucene query parser.

1. Select **Search explorer** on the command bar.

2. From **Index**, choose "hotels-sample-index".

3. In the search bar, paste in a query string from the examples below and select **Search**.

Run more example queries

All of the queries in this section are designed for **Search Explorer** and the Hotels sample index. Results are returned as verbose JSON documents. All fields marked as "retrievable" in the index can appear in results. For more information about queries, see Querying in Azure Cognitive Search.

Query	Description
`search=spa`	Simple full text query with top N results. The **`search=`** parameter is used for keyword search, in this case, returning hotel data for those containing *spa* in any searchable field in the document.
`search=beach &$filter=Rating gt 4`	Filtered query. In this case, ratings greater than 4.
`search=spa &$select=HotelName,Description,Tags &$count=true &$top=10`	Parameterized query. The **`&`** symbol is used to append search parameters, which can be specified in any order. **`$select`** parameter returns a subset of fields for more concise search results. **`$count=true`** parameter returns the total count of all documents that match the query. **`$top=10`** returns the highest ranked 10 documents out of the total. By default, Azure Cognitive Search returns the first 50 best matches. You can increase or decrease the amount using this parameter.
`search=* &facet=Category &$top=2`	Facet query, used to return an aggregated count of documents that match a facet value you provide. On an empty or unqualified search, all documents are represented. In the hotels index, the Category field is marked as "facetable".
`search=spa &facet=Rating`	Facet on numeric values. This query is facet for rating, on a text search for "spa". The term "Rating" can be specified as a facet because the field is marked as retrievable, filterable, and facetable in the index, and its numeric values (1 through 5) are suitable for grouping results by each value.
`search=beach &highlight=Description &$select=HotelName, Description, Category, Tags`	Hit highlighting. The term "beach" will be highlighted when it appears in the "Description" field.

Query	Description
`search=seatle` followed by `search=seatle~ &queryType=full`	Fuzzy search. By default, misspelled query terms, like *seatle* for "Seattle", fail to return matches in typical search. The first example returns no results. Adding **`queryType=full`** invokes the full Lucene query parser, which supports the `~` operand for fuzzy search.
`$filter=geo.distance(Location, geography'POINT(-122.12 47.67)') le 5 &search=* &$select=HotelName, Address/City, Address/StateProvince &$count=true`	Geospatial search. The example query filters all results for positional data, where results are less than 5 kilometers from a given point, as specified by latitude and longitude coordinates (this example uses Redmond, Washington as the point of origin).

Takeaways

This quickstart provided a quick introduction to Azure Cognitive Search using the Azure portal.

You learned how to create a search index using the **Import data** wizard. You created your first indexer and learned the basic workflow for index design. See Import data wizard in Azure Cognitive Search for more information about the wizard's benefits and limitations.

Using the **Search explorer** in the Azure portal, you learned some basic query syntax through hands-on examples that demonstrated key capabilities such as filters, hit highlighting, fuzzy search, and geospatial search.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that the limit is three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

Use a portal wizard to generate a ready-to-use web app that runs in a browser. You can try out this wizard on the small index you just created, or use one of the built-in sample data sets for a richer search experience.

Create a demo app in the portal

Quickstart: Create a demo app in the portal (Azure Cognitive Search)

Article • 10/13/2022 • 5 minutes to read

In this quickstart, you'll use the Azure portal's **Create demo app** wizard to generate a downloadable, "localhost"-style web app that runs in a browser. Depending on its configuration, the generated app is operational on first use, with a live read-only connection to an index on your search service. A default app can include a search bar, results area, sidebar filters, and typeahead support.

A demo app can help you visualize how an index will function in a client app, but it isn't intended for production scenarios. Production apps should include security, error handling, and hosting logic that the demo app doesn't provide.

Prerequisites

Before you begin, have the following prerequisites in place:

- An Azure account with an active subscription. Create an account for free .
- An Azure Cognitive Search service. Create a service or find an existing service under your current subscription. You can use a free service for this quickstart.
- Microsoft Edge (latest version) or Google Chrome.
- A search index to use as the basis of your generated application.

  This quickstart uses the built-in Real Estate sample data and index because it has thumbnail images (the wizard supports adding images to the results page). To create the index used in this exercise, run the **Import data** wizard, choosing the *realestate-us-sample* data source.

When the index is ready to use, move on to the next step.

Start the wizard

1. Sign in to the Azure portal with your Azure account.
2. Find your search service and on the Overview page, from the links on the middle of the page, select **Indexes**.
3. Choose *realestate-us-sample-index* from the list of existing indexes.
4. On the index page, at the top, select **Create demo app (preview)** to start the wizard.
5. On the first wizard page, select **Enable Cross Origin Resource Sharing (CORS)** to add CORS support to your index definition. This step is optional, but your local web app won't connect to the remote index without it.

Configure search results

The wizard provides a basic layout for rendered search results that includes space for a thumbnail image, a title, and description. Backing each of these elements is a field in your index that provides the data.

1. In Thumbnail, choose the *thumbnail* field in the *realestate-us-sample* index. This sample happens to include image thumbnails in the form of URL-addressed images stored in a field called *thumbnail*. If your index doesn't have images, leave this field blank.
2. In Title, choose a field that conveys the uniqueness of each document. In this sample, the listing ID is a reasonable selection.

3. In Description, choose a field that provides details that might help someone decide whether to click through to that particular document.

Add a sidebar

The search service supports faceted navigation, which is often rendered as a sidebar. Facets are based on filterable and facetable fields, as expressed in the index schema.

In Azure Cognitive Search, faceted navigation is a cumulative filtering experience. Within a category, selecting multiple filters expands the results (for example, selecting Seattle and Bellevue within City). Across categories, selecting multiple filters narrows results.

> **Tip**
>
> You can view the full index schema in the portal. Look for the **Index definition (JSON)** link in each index's overview page. Fields that qualify for faceted navigation have "filterable: true" and "facetable: true" attributes.

1. In the wizard, select the **Sidebar** tab at the top of the page. You'll see a list of all fields that are attributed as filterable and facetable in the index.

2. Accept the current selection of faceted fields and continue to the next page.

Add typeahead

Typeahead functionality is available in the form of autocomplete and query suggestions. The wizard supports query suggestions. Based on keystroke inputs provided by the user, the search service returns a list of "completed" query strings that can be selected as the input.

Suggestions are enabled on specific field definitions. The wizard gives you options for configuring how much information is included in a suggestion.

The following screenshot shows options in the wizard, juxtaposed with a rendered page in the app. You can see how field selections are used, and how "Show Field Name" is used to include or exclude labeling within the suggestion.

Add suggestions

Suggestions refer to automated query prompts that are attached to the search box. Cognitive Search supports two: *autocompletion* of a partially entered search term, and *suggestions* for a dropdown list of potential matching documents based.

The wizard supports suggestions, and the fields that can provide suggested results are derived from a Suggesters construct in the index:

JSON

```
"suggesters": [
  {
    "name": "sg",
    "searchMode": "analyzingInfixMatching",
    "sourceFields": [
      "number",
      "street",
      "city",
      "region",
      "postCode",
      "tags"
    ]
```

1. In the wizard, select the **Suggestions** tab at the top of the page. You'll see a list of all fields that are designated in the index schema as suggestion providers.

2. Accept the current selection and continue to the next page.

Create, download and execute

1. Select **Create demo app** at the bottom of the page to generate the HTML file.

2. When prompted, select **Download your app** to download the file.

3. Open the file and click the Search button. This action executes a query, which can be an empty query (`*`) that returns an arbitrary result set. The page should look similar to the following screenshot. Enter a term and use filters to narrow results.

The underlying index is composed of fictitious, generated data that has been duplicated across documents, and descriptions sometimes don't match the image. You can expect a more cohesive experience when you create an app based on your own indexes.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that it's limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

The demo app is useful for prototyping because you can simulate an end-user experience without having to write JavaScript or front-end code, but as you get closer to proof-of-concept in your own project, review one of the end-to-end code samples that is closer facsimile of a real-word app:

Add search to web apps

Quickstart: Create an Azure Cognitive Search skillset in the Azure portal

Article • 10/13/2022 • 7 minutes to read

In this quickstart, you'll learn how AI enrichment in Azure Cognitive Search adds Optical Character Recognition (OCR), image analysis, language detection, text translation, and entity recognition to create text-searchable content in a search index.

You'll run the **Import data** wizard in the Azure portal to apply skills that transform and enrich content during indexing. Output is a searchable index containing AI-generated image text, captions, and entities. Enriched content is queryable in the portal using Search explorer.

To prepare, you'll create a few resources and upload sample files before running the wizard.

Prerequisites

Before you begin, have the following prerequisites in place:

- An Azure account with an active subscription. Create an account for free.
- Azure Cognitive Search. Create a service or find an existing service. You can use a free service for this quickstart.
- Azure Storage account with Blob Storage.

> **Note**
>
> This quickstart uses **Cognitive Services** for the AI. Because the workload is so small, Cognitive Services is tapped behind the scenes for free processing for up to 20 transactions. You can complete this exercise without having to create a Cognitive Services resource.

Set up your data

In the following steps, set up a blob container in Azure Storage to store heterogeneous content files.

1. Download sample data consisting of a small file set of different types. Unzip the files.

2. Sign in to the Azure portal with your Azure account.

3. Create an Azure Storage account or find an existing account.

   - Choose the same region as Azure Cognitive Search to avoid bandwidth charges.

   - Choose the StorageV2 (general purpose V2).

4. In Azure portal, open your Azure Storage page and create a container. You can use the default public access level.

5. In Container, select **Upload** to upload the sample files you downloaded in the first step. Notice that you have a wide range of content types, including images and application files that aren't full text searchable in their native formats.

NAME	LAST MODIFIED	BLOB TYPE	CONTENT TYPE
10-K-FY16.html	2018-04-02T21:10:35.000Z	Block Blob	text/html
5074.clip_image002_6FE27E85.png	2018-04-02T21:10:39.000Z	Block Blob	image/png
Cognitive Searvices and Content Intelligence.pptx	2018-04-02T21:10:50.000Z	Block Blob	application/vnd.openxmlformats-officedo
Cognitive Services and Bots (spanish).pdf	2018-04-02T21:10:41.000Z	Block Blob	application/pdf
guthrie.jpg	2018-04-02T21:10:23.000Z	Block Blob	image/jpeg
MSFT_cloud_architecture_contoso.pdf	2018-04-02T21:10:35.000Z	Block Blob	application/pdf
MSFT_FY17_10K.docx	2018-04-02T21:10:36.000Z	Block Blob	application/vnd.openxmlformats-officedo
NYSE_LNKD_2015.PDF	2018-04-02T21:10:35.000Z	Block Blob	application/pdf
redshirt.jpg	2018-04-02T21:10:31.000Z	Block Blob	image/jpeg
satyanadellalinux.jpg	2018-04-02T21:10:33.000Z	Block Blob	image/jpeg
satyasletter.txt	2018-04-02T21:10:22.000Z	Block Blob	text/plain

You're now ready to move on the Import data wizard.

Run the Import data wizard

1. Sign in to the Azure portal with your Azure account.

2. Find your search service and on the Overview page, select **Import data** on the command bar to set up cognitive enrichment in four steps.

Step 1 - Create a data source

1. In **Connect to your data**, choose **Azure Blob Storage**.

2. Choose an existing connection to the storage account and select the container you created. Give the data source a name, and use default values for the rest.

Continue to the next page.

Step 2 - Add cognitive skills

Next, configure AI enrichment to invoke OCR, image analysis, and natural language processing.

1. For this quickstart, we're using the **Free** Cognitive Services resource. The sample data consists of 14 files, so the free allotment of 20 transaction on Cognitive Services is sufficient for this quickstart.

2. Expand **Add enrichments** and make six selections.

   Enable OCR to add image analysis skills to wizard page.

   Choose entity recognition (people, organizations, locations) and image analysis skills (tags, captions).

Continue to the next page.

Step 3 - Configure the index

An index contains your searchable content and the **Import data** wizard can usually create the schema for you by sampling the data source. In this step, review the generated schema and potentially revise any settings. Below is the default schema created for the demo Blob data set.

For this quickstart, the wizard does a good job setting reasonable defaults:

- Default fields are based on metadata properties for existing blobs, plus the new fields for the enrichment output (for example, `people`, `organizations`, `locations`). Data types are inferred from metadata and by data sampling.

- Default document key is *metadata_storage_path* (selected because the field contains unique values).

- Default attributes are **Retrievable** and **Searchable**. **Searchable** allows full text search a field. **Retrievable** means field values can be returned in results. The wizard assumes you want these fields to be retrievable and searchable because you created them via a skillset. Select **Filterable** if you want to use fields in a filter expression.

Marking a field as **Retrievable** doesn't mean that the field *must* be present in the search results. You can control search results composition by using the **$select** query parameter to specify which fields to include.

Continue to the next page.

Step 4 - Configure the indexer

The indexer drives the indexing process. It specifies the data source name, a target index, and frequency of execution. The **Import data** wizard creates several objects, including an indexer that you can reset and run repeatedly.

1. In the **Indexer** page, you can accept the default name and select **Once** to run it immediately.

2. Select **Submit** to create and simultaneously run the indexer.

Monitor status

Cognitive skills indexing takes longer to complete than typical text-based indexing, especially OCR and image analysis. To monitor progress, go to the Overview page and select **Indexers** in the middle of page.

To check details about execution status, select an indexer from the list, and then select **Success** (or **Failed**) to view execution details.

In this demo, there's one warning: `"Could not execute skill because one or more skill input was invalid." It tells you that a PNG file in the data source doesn't provide a text input to Entity Recognition. This warning occurs because the upstream OCR skill didn't recognize any text in the image, and thus couldn't provide a text input to the downstream Entity Recognition skill.

Warnings are common in skillset execution. As you become familiar with how skills iterate over your data, you'll begin to notice patterns and learn which warnings are safe to ignore.

Query in Search explorer

After an index is created, run queries in **Search explorer** to return results.

1. On the search service dashboard page, select **Search explorer** on the command bar.

2. Select **Change Index** at the top to select the index you created.

3. Enter a search string to query the index, such as `search=Satya Nadella&$select=people,organizations,locations&$count=true`.

Results are returned as verbose JSON, which can be hard to read, especially in large documents. Some tips for searching in this tool include the following techniques:

- Append `$select` to limit the fields returned in results.

- Use CTRL-F to search within the JSON for specific properties or terms.

Query strings are case-sensitive so if you get an "unknown field" message, check **Fields** or **Index Definition (JSON)** to verify name and case.

Takeaways

You've now created your first skillset and learned important concepts useful for prototyping an enriched search solution using your own data.

Some key concepts that we hope you picked up include the dependency on Azure data sources. A skillset is bound to an indexer, and indexers are Azure and source-specific. Although this quickstart uses Azure Blob Storage, other Azure data sources are possible. For more information, see Indexers in Azure Cognitive Search.

Another important concept is that skills operate over content types, and when working with heterogeneous content, some inputs will be skipped. Also, large files or fields might exceed the indexer limits of your service tier. It's normal to see warnings when these events occur.

Output is directed to a search index, and there's a mapping between name-value pairs created during indexing and individual fields in your index. Internally, the portal sets up annotations and defines a skillset, establishing the order of operations and general flow. These steps are hidden in the portal, but when you start writing code, these concepts become important.

Finally, you learned that can verify content by querying the index. In the end, what Azure Cognitive Search provides is a searchable index, which you can query using either the simple or fully extended query syntax. An index containing enriched fields is like any other. If you want to incorporate standard or custom analyzers, scoring profiles, synonyms, faceted navigation, geo-search, or any other Azure Cognitive Search feature, you can certainly do so.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

You can create skillsets using the portal, .NET SDK, or REST API. To further your knowledge, try the REST API using Postman and more sample data.

Tutorial: Extract text and structure from JSON blobs using REST APIs

Quickstart: Create a knowledge store in the Azure portal

Article • 10/13/2022 • 6 minutes to read

In this quickstart, you'll create a knowledge store that serves as a repository for output created from an AI enrichment pipeline. A knowledge store makes enriched content available in Azure Storage for downstream apps and workloads, for other work besides full text search.

First, you'll set up some sample data. Then, you'll run the **Import data** wizard to create an enrichment pipeline that also generates a knowledge store. The knowledge store will contain original text content pulled from the data source (customer reviews of a hotel), plus AI-generated content that includes a sentiment label, key phrase extraction, and text translation of non-English customer comments.

Prerequisites

Before you begin, have the following prerequisites in place:

- An Azure account with an active subscription. Create an account for free.
- Azure Cognitive Search. Create a service or find an existing service in your account. You can use a free service for this quickstart.
- Azure Storage. Create an account or find an existing account. The account type must be **StorageV2 (general purpose V2)**.
- Sample data hosted in Azure Storage:

  Download HotelReviews_Free.csv. This CSV contains 19 pieces of customer feedback about a single hotel (originates from Kaggle.com). The file is in a repo with other sample data. If you don't want the whole repo, copy the raw content and paste it into a spreadsheet app on your device.

  Upload the file to a blob container in Azure Storage.

This quickstart also uses Cognitive Services for AI enrichment. Because the workload is so small, Cognitive Services is tapped behind the scenes for free processing for up to 20 transactions. This means that you can complete this exercise without having to create an extra Cognitive Services resource.

Start the wizard

1. Sign in to the Azure portal with your Azure account.

2. Find your search service and on the Overview page, select **Import data** on the command bar to create a knowledge store in four steps.

Step 1: Create a data source

Because the data is multiple rows in one CSV file, set the *parsing mode* to get one search document for each row.

1. In **Connect to your data**, choose **Azure Blob Storage**.

2. For the **Name**, enter "hotel-reviews-ds".

3. For **Data to extract**, choose **Content and Metadata**.

4. For **Parsing mode**, select **Delimited text**, and then select the **First Line Contains Header** checkbox. Make sure the **Delimiter character** is a comma (,).

5. In **Connection String**, choose an existing connection if the storage account is in the same subscription. Otherwise, paste in a connection string to your Azure Storage account.

   A connection string can be full access, having the following format: `DefaultEndpointsProtocol=https;AccountName=<YOUR-ACCOUNT-NAME>;AccountKey=<YOUR-ACCOUNT-KEY>;EndpointSuffix=core.windows.net`

   Or, a connection string can reference a managed identity, assuming it's configured and assigned a role in Azure Storage: `ResourceId=/subscriptions/{YOUR-SUBSCRIPTION-ID}/resourceGroups/{YOUR-RESOURCE-GROUP-NAME}/providers/Microsoft.Storage/storageAccounts/{YOUR-ACCOUNT-NAME};`

6. In **Containers**, enter the name of the blob container holding the data ("hotel-reviews").

   Your page should look similar to the following screenshot.

7. Continue to the next page.

Step 2: Add skills

In this wizard step, add skills for AI enrichment. The source data consists of customer reviews in English and French. Skills that are relevant for this data set include key phrase extraction, sentiment detection, and text translation. In a later step, these enrichments will be "projected" into a knowledge store as Azure tables.

1. Expand **Attach Cognitive Services**. **Free (Limited enrichments)** is selected by default. You can use this resource because the number of records in HotelReviews-Free.csv is 19 and this free resource allows up to 20 transactions a day.

2. Expand **Add enrichments**.

3. For **Skillset name**, enter "hotel-reviews-ss".

4. For **Source data field**, select **reviews_text**.

5. For **Enrichment granularity level**, select **Pages (5000 characters chunks)**.

6. For **Text Cognitive Skills**, select the following skills:

   - **Extract key phrases**
   - **Translate text**

- **Language detection**
- **Detect sentiment**

Your page should look like the following screenshot:

7. Scroll down and expand **Save enrichments to knowledge store**.

8. Select **Choose an existing connection** and then select an Azure Storage account. The Containers page will appear so that you can create a container for projections. We recommend adopting a prefix naming convention, such as "kstore-hotel-reviews" to distinguish between source content and knowledge store content.

9. Returning to the Import data wizard, select the following **Azure table projections**. The wizard always offers the **Documents** projection. Other projections will be offered depending on the skills you select (such as **Key phrases**), or the enrichment granularity (**Pages**):

   - **Documents**
   - **Pages**
   - **Key phrases**

   The following screenshot shows the table projection selections in the wizard.

10. Continue to the next page.

Step 3: Configure the index

In this wizard step, configure an index for optional full-text search queries. The wizard will sample your data source to infer fields and data types. You only need to select the attributes for your desired behavior. For example, the **Retrievable** attribute will allow the search service to return a field value while the **Searchable** will enable full text search on the field.

1. For **Index name**, enter "hotel-reviews-idx".

2. For attributes, accept the default selections: **Retrievable** and **Searchable** for the new fields that the pipeline is creating.

   Your index should look similar to the following image. Because the list is long, not all fields are visible in the image.

3. Continue to the next page.

Step 4: Configure and run the indexer

In this wizard step, configure an indexer that will pull together the data source, skillset, and the index you defined in the previous wizard steps.

1. For **Name**, enter "hotel-reviews-idxr".
2. For **Schedule**, keep the default **Once**.
3. Select **Submit** to run the indexer. Data extraction, indexing, application of cognitive skills all happen in this step.

Step 5: Check status

In the **Overview** page, open the **Indexers** tab in the middle of the page, and then select **hotels-reviews-idxr**. Within a minute or two, status should progress from "In progress" to "Success" with zero errors and warnings.

Check tables in Azure portal

1. In the Azure portal, open the Storage account used to create the knowledge store.
2. In the storage account's left navigation pane, select **Storage browser (preview)** to view the new tables.

   You should see three tables, one for each projection that was offered in the "Save enrichments" section of the "Add enrichments" page.

   - "hotelReviewssDocuments" contains all of the first-level nodes of a document's enrichment tree that aren't collections.
   - "hotelReviewssKeyPhrases" contains a long list of just the key phrases extracted from all reviews. Skills that output collections (arrays), such as key phrases and entities, will have output sent to a standalone table.
   - "hotelReviewssPages" contains enriched fields created over each page that was split from the document. In this skillset and data source, page-level enrichments consisting of sentiment labels and translated text. A pages table (or a sentences table if you specify that particular level of granularity) is created when you choose "pages" granularity in the skillset definition.

All of these tables contain ID columns to support table relationships in other tools and apps. When you open a table, scroll past these fields to view the content fields added by the pipeline.

In this quickstart, the table for "hotelReviewssPages" should look similar to the following screenshot:

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

> **Tip**
>
> If you want to repeat this exercise or try a different AI enrichment walkthrough, delete the **hotel-reviews-idxr** indexer and the related objects to recreate them. Deleting the indexer resets the free daily transaction counter to zero.

Next steps

Now that you've been introduced to a knowledge store, take a closer look at each step by switching over to the REST API walkthrough. Tasks that the wizard handled internally are explained in the REST walkthrough.

Create a knowledge store using REST and Postman

Quickstart: Use Search explorer to run queries in the portal

Article • 11/16/2022 • 5 minutes to read

In this quickstart, you'll learn how to use **Search explorer**, a built-in query tool in the Azure portal used for running queries against a search index in Azure Cognitive Search. This tool makes it easy to learn query syntax, test a query or filter expression, or confirm data refresh by checking whether new content exists in the index.

This quickstart uses an existing index to demonstrate Search explorer.

Prerequisites

Before you begin, have the following prerequisites in place:

- An Azure account with an active subscription. Create an account for free.
- An Azure Cognitive Search service. Create a service or find an existing service under your current subscription. You can use a free service for this quickstart.
- The *realestate-us-sample-index* is used for this quickstart. To create the index, use the **Import data wizard**, choose the sample data, and step through the wizard using all of the default values.

Start Search explorer

1. In the Azure portal, open the search overview page from the dashboard or find your service.

2. Open Search explorer from the command bar:

Or use the embedded **Search explorer** tab on an open index:

Unspecified query

In Search explorer, requests are formulated using the Search REST API, with responses returned as verbose JSON documents.

For a first look at content, execute an empty search by clicking **Search** with no terms provided. An empty search is useful as a first query because it returns entire documents so that you can review document composition. On an empty search, there's no search rank and documents are returned in arbitrary order (`"@search.score": 1` for all documents). By default, 50 documents are returned in a search request.

Equivalent syntax for an empty search is `*` or `search=*`.

HTTP

```
search=*
```

Results

Query string

Examples: *, $top=10, $top=10&$skip=10&search=*

API version

2020-06-30

Search

Request URL

https://demo-search-svc.search.windows.net/indexes/realestate-us-sample-index/docs?api-version=2020-06-30&search=*

Results


```
1  {
2      "@odata.context": "https://demo-search-svc.search.windows.net/indexes('realestate-us-sample-index')/$metadata#docs(*)",
3      "value": [
4          {
5              "@search.score": 1,
6              "listingId": "OTM4NjM0MA2",
7              "beds": 5,
8              "baths": 4,
9              "description": "This is a condo and is well maintained.  Enjoy gleaming hardwood floors located close to parks and features stainless steel appliances, a generous backyard and a wet bar.",
10             "description_de": "Dies ist eine Eigentumswohnung und ist sehr gepflegt.  Genießen Sie glänzenden Holzböden Parks in der Nähe und Funktionen-Edelstahl-Geräte, ein großzügiger Garten and eine wet-bar.",
```


Free text search

Free-form queries, with or without operators, are useful for simulating user-defined queries sent from a custom app to Azure Cognitive Search. Only those fields attributed as **Searchable** in the index definition are scanned for matches.

Notice that when you provide search criteria, such as query terms or expressions, search rank comes into play. The following example illustrates a free text search. The "@search.score" is a relevance score computed for the match using the default scoring algorithm.

HTTP

```
Seattle apartment "Lake Washington" miele OR thermador appliance
```

Results

You can use Ctrl-F to search within results for specific terms of interest.

Query string

Seattle apartment "Lake Washington" miele OR thermador appliance

API version

2020-06-30

Search

Request URL

https://demo-search-svc.search.windows.net/indexes/realestate-us-sample-index/docs?api-version=2020-06-30&search=Seattle%20apartment%20%22Lake%20Washing...

Results

thermador Aa Ab .* 1 of 153


```
",
    "value": [
        {
            "@search.score": 42.890236,
            "listingId": "OTM4MzExMw2",
            "beds": 4,
            "baths": 4,
            "description": "This is a bungalow and is move in condition.  This home provides a large lot located close to parks and features thermador appliances, a generous backyard and a guest house.",
            "description_de": "Dies ist ein bungalow und ist in einem Zustand zu bewegen.  Dieses Haus bietet eine große Menge Parks in der Nähe und Funktionen Thermador Geräte, ein großzügiger Garten and ein Gästehaus.",
```


Count of matching documents

Add **$count=true** to get the number of matches found in an index. On an empty search, count is the total number of documents in the index. On a qualified search, it's the number of documents matching the query input. Recall that the service returns the top 50 matches by default, so the count might indicate more matches in the index than what's returned in the results.

HTTP

```
$count=true
```

Results

Limit fields in search results

Add **$select** to limit results to the explicitly named fields for more readable output in **Search explorer**. To keep the previously mentioned parameters in the query, use **&** to separate each parameter.

HTTP

```
search=seattle
condo&$select=listingId,beds,baths,description,street,city,price&$count=true
```

Results

Query string

search=seattle condo&$select=listingId,beds,baths,description,street,city,price&$count=true

API version: 2020-06-30 Search

Request URL

https://demo-search-svc.search.windows.net/indexes/realestate-us-sample-index/docs?api-version=2020-06-30&search=seattle%20condo&%24select=listingId%2Cbeds...

Results


```
{
    "@odata.context": "https://demo-search-svc.search.windows.net/indexes('realestate-us-sample-index')/$metadata#docs(*)",
    "@odata.count": 2056,
    "value": [
        {
            "@search.score": 13.861284,
            "listingId": "OTM4Mjg2Ng2",
            "beds": 2,
            "baths": 2,
            "description": "This is a condo and is a beautiful home.  Enjoy oceanfrontage located close to a river and features an office, a garden and granite countertops.",
            "street": "24th Avenue South",
            "city": "Seattle",
            "price": 637632
```


Return next batch of results

Azure Cognitive Search returns the top 50 matches based on the search rank. To get the next set of matching documents, append **$top=100,&$skip=50** to increase the result set to 100 documents (default is 50, maximum is 1000), skipping the first 50 documents. You can check the document key (listingID) to identify a document.

Recall that you need to provide search criteria, such as a query term or expression, to get ranked results. Notice that search scores decrease the deeper you reach into search results.

HTTP

```
search=seattle
condo&$select=listingId,beds,baths,description,street,city,price&$count=true
&$top=100&$skip=50
```

Results

Query string

search=seattle condo&$select=listingId,beds,baths,description,street,city,price&$count=true&$top=100&$skip=50

API version: 2020-06-30 Search

Request URL

https://demo-search-svc.search.windows.net/indexes/realestate-us-sample-index/docs?api-version=2020-06-30&search=seattle%20condo&%24select=listingId%2Cbeds...

Results


```
{
    "@odata.context": "https://demo-search-svc.search.windows.net/indexes('realestate-us-sample-index')/$metadata#docs(*)",
    "@odata.count": 2056,
    "value": [
        {
            "@search.score": 12.719713,
            "listingId": "OTM4Mjc2OQ2",
            "beds": 5,
            "baths": 3,
            "description": "This is a condo and is a short sale.  Enjoy lake front property located within walking distance of a beach and features a lap pool, a garden and a covered front porch.",
            "street": "13th Avenue South",
            "city": "Seattle",
            "price": 2702160
        },
```

Filter expressions (greater than, less than, equal to)

Use the **$filter** parameter when you want to specify precise criteria rather than free text search. The field must be attributed as **Filterable** in the index. This example searches for bedrooms greater than 3:

HTTP

```
search=seattle condo&$filter=beds gt 3&$count=true
```

Results

Sorting results

Add **$orderby** to sort results by another field besides search score. The field must be attributed as **Sortable** in the index. An example expression you can use to test this out is:

HTTP

```
search=seattle condo&$select=listingId,beds,price&$filter=beds gt
3&$count=true&$orderby=price asc
```

Results

Both **$filter** and **$orderby** expressions are OData constructions. For more information, see Filter OData syntax.

Takeaways

In this quickstart, you used **Search explorer** to query an index using the REST API.

- Results are returned as verbose JSON documents so that you can view document construction and content, in entirety. The **$select** parameter in a query expression can limit which fields are returned.
- Search results are composed of all fields marked as **Retrievable** in the index. To view field attributes in the portal, select *realestate-us-sample* in the **Indexes** list on the search overview page, and then open the **Fields** tab.
- Keyword search, similar to what you might enter in a commercial web browser, are useful for testing an end-user experience. For example, assuming the built-in real estate sample index, you could enter "Seattle apartments lake washington", and then you can use Ctrl-F to find terms within the search results.
- Query and filter expressions are articulated in a syntax implemented by Azure Cognitive Search. The default is a simple syntax, but you can optionally use full Lucene for more powerful queries. Filter expressions are articulated in an OData syntax.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can

cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

To learn more about query structures and syntax, use Postman or an equivalent tool to create query expressions that use more parts of the API. The Search REST API is especially helpful for learning and exploration.

Create a basic query in Postman

Additional resources

Documentation

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Run or reset indexers - Azure Cognitive Search

Run indexers in full, or reset an indexer, skills, or individual documents to refresh all or part of a search index or knowledge store.

Create Data Source (Azure Cognitive Search REST API)

Data source connection information used by an Azure Cognitive Search indexer when crawling external data sources.

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Quickstart: Create a search index using REST APIs - Azure Cognitive Search

In this REST API quickstart, learn how to call the Azure Cognitive Search REST APIs using Postman.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Show 5 more

Training

Learning path

Microsoft Search fundamentals - Training

Learn about Microsoft Search including where users can search, the answers and results they'll see, and how you can tailor the search experience for your organization.

Quickstart: Create a search index using the Azure.Search.Documents client library

Article • 04/09/2023 • 15 minutes to read

Learn how to use the Azure.Search.Documents (version 11) client library to create a .NET Core console application in C# that creates, loads, and queries a search index.

You can download the source code to start with a finished project or follow the steps in this article to create your own.

> **Note**
>
> Looking for an earlier version? See **Create a search index using Microsoft.Azure.Search v10** instead.

Prerequisites

Before you begin, have the following tools and services:

- An Azure account with an active subscription. Create an account for free.
- An Azure Cognitive Search service. Create a service or find an existing service. You can use a free service for this quickstart.
- Visual Studio, any edition. Sample code was tested on the free Community edition of Visual Studio 2019.

When setting up your project, you'll download the Azure.Search.Documents NuGet package.

Azure SDK for .NET conforms to .NET Standard 2.0, which means .NET Framework 4.6.1 and .NET Core 2.1 as minimum requirements.

Set up your project

Assemble service connection information, and then start Visual Studio to create a new Console App project that can run on. Select NET Core 3.1 for the run time.

Copy a key and endpoint

Calls to the service require a URL endpoint and an access key on every request. As a first step, find the API key and URL to add to your project. You'll specify both values when creating the client in a later step.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service, required if you're creating or deleting objects. There are two interchangeable primary and secondary keys. You can use either one.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Install the NuGet package

After the project is created, add the client library. The Azure.Search.Documents package consists of one client library that provides all of the APIs used to work with a search service in .NET.

1. Start Visual Studio and create a .NET Core console application.

2. In **Tools** > **NuGet Package Manager**, select **Manage NuGet Packages for Solution...**.

3. Select **Browse**.

4. Search for `Azure.Search.Documents` and select version 11.0 or later.

5. Select **Install** on the right to add the assembly to your project and solution.

Create a search client

1. In **Program.cs**, change the namespace to `AzureSearch.SDK.Quickstart.v11` and then add the following `using` directives.

```C#
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.Models;
```

2. Create two clients: SearchIndexClient creates the index, and SearchClient loads and queries an existing index. Both need the service endpoint and an admin API key for authentication with create/delete rights.

```C#
 static void Main(string[] args)
 {
     string serviceName = "<your-search-service-name>";
     string apiKey = "<your-search-service-admin-api-key>";
     string indexName = "hotels-quickstart";

     // Create a SearchIndexClient to send create/delete index commands
     Uri serviceEndpoint = new
Uri($"https://{serviceName}.search.windows.net/");
     AzureKeyCredential credential = new AzureKeyCredential(apiKey);
     SearchIndexClient adminClient = new
SearchIndexClient(serviceEndpoint, credential);

     // Create a SearchClient to load and query documents
     SearchClient srchclient = new SearchClient(serviceEndpoint,
indexName, credential);
     . . .
 }
```

1 - Create an index

This quickstart builds a Hotels index that you'll load with hotel data and execute queries against. In this step, define the fields in the index. Each field definition includes a name, data type, and attributes that determine how the field is used.

In this example, synchronous methods of the Azure.Search.Documents library are used for simplicity and readability. However, for production scenarios, you should use asynchronous methods to keep your app scalable and responsive. For example, you would use CreateIndexAsync instead of CreateIndex.

1. Add an empty class definition to your project: **Hotel.cs**

2. Copy the following code into **Hotel.cs** to define the structure of a hotel document. Attributes on the field determine how it's used in an application. For example, the `IsFilterable` attribute must be assigned to every field that supports a filter expression.

C#

```
using System;
using System.Text.Json.Serialization;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;

namespace AzureSearch.Quickstart
{
    public partial class Hotel
    {
        [SimpleField(IsKey = true, IsFilterable = true)]
        public string HotelId { get; set; }

        [SearchableField(IsSortable = true)]
        public string HotelName { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.EnLucene)]
        public string Description { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.FrLucene)]
        [JsonPropertyName("Description_fr")]
        public string DescriptionFr { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string Category { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string[] Tags { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
```

```
        public bool? ParkingIncluded { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public DateTimeOffset? LastRenovationDate { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public double? Rating { get; set; }

        [SearchableField]
        public Address Address { get; set; }
    }
}
```

In the Azure.Search.Documents client library, you can use SearchableField and SimpleField to streamline field definitions. Both are derivatives of a SearchField and can potentially simplify your code:

- `SimpleField` can be any data type, is always non-searchable (it's ignored for full text search queries), and is retrievable (it's not hidden). Other attributes are off by default, but can be enabled. You might use a `SimpleField` for document IDs or fields used only in filters, facets, or scoring profiles. If so, be sure to apply any attributes that are necessary for the scenario, such as `IsKey = true` for a document ID. For more information, see SimpleFieldAttribute.cs in source code.

- `SearchableField` must be a string, and is always searchable and retrievable. Other attributes are off by default, but can be enabled. Because this field type is searchable, it supports synonyms and the full complement of analyzer properties. For more information, see the SearchableFieldAttribute.cs in source code.

Whether you use the basic `SearchField` API or either one of the helper models, you must explicitly enable filter, facet, and sort attributes. For example, IsFilterable, IsSortable, and IsFacetable must be explicitly attributed, as shown in the sample above.

3. Add a second empty class definition to your project: **Address.cs**. Copy the following code into the class.

C#

```
using Azure.Search.Documents.Indexes;

 namespace AzureSearch.Quickstart
```

```
{
    public partial class Address
    {
        [SearchableField(IsFilterable = true)]
        public string StreetAddress { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string City { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string StateProvince { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string PostalCode { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string Country { get; set; }
    }
}
```

4. Create two more classes: **Hotel.Methods.cs** and **Address.Methods.cs** for ToString() overrides. These classes are used to render search results in the console output. The contents of these classes aren't provided in this article, but you can copy the code from files in GitHub.

5. In **Program.cs**, create a SearchIndex object, and then call the CreateIndex method to express the index in your search service. The index also includes a SearchSuggester to enable autocomplete on the specified fields.

C#

```
// Create hotels-quickstart index
private static void CreateIndex(string indexName, SearchIndexClient
adminClient)
{
    FieldBuilder fieldBuilder = new FieldBuilder();
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var definition = new SearchIndex(indexName, searchFields);

    var suggester = new SearchSuggester("sg", new[] { "HotelName",
"Category", "Address/City", "Address/StateProvince" });
    definition.Suggesters.Add(suggester);

    adminClient.CreateOrUpdateIndex(definition);
}
```

2 - Load documents

Azure Cognitive Search searches over content stored in the service. In this step, you'll load JSON documents that conform to the hotel index you just created.

In Azure Cognitive Search, search documents are data structures that are both inputs to indexing and outputs from queries. As obtained from an external data source, document inputs might be rows in a database, blobs in Blob storage, or JSON documents on disk. In this example, we're taking a shortcut and embedding JSON documents for four hotels in the code itself.

When uploading documents, you must use an IndexDocumentsBatch object. An `IndexDocumentsBatch` object contains a collection of Actions, each of which contains a document and a property telling Azure Cognitive Search what action to perform (upload, merge, delete, and mergeOrUpload).

1. In **Program.cs**, create an array of documents and index actions, and then pass the array to `IndexDocumentsBatch`. The documents below conform to the hotels-quickstart index, as defined by the hotel class.

C#

```
// Upload documents in a single Upload request.
private static void UploadDocuments(SearchClient searchClient)
{
    IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                HotelName = "Secret Point Motel",
                Description = "The hotel is ideally located on the main
commercial artery of the city in the heart of New York. A few minutes
away is Time's Square and the historic centre of the city, as well as
other places of interest that make New York one of America's most
attractive and cosmopolitan cities.",
                DescriptionFr = "L'hôtel est idéalement situé sur la
principale artère commerciale de la ville en plein cœur de New York. A
quelques minutes se trouve la place du temps et le centre historique de
la ville, ainsi que d'autres lieux d'intérêt qui font de New York l'une
des villes les plus attractives et cosmopolites de l'Amérique.",
                Category = "Boutique",
                Tags = new[] { "pool", "air conditioning", "concierge"
},
                ParkingIncluded = false,
                LastRenovationDate = new DateTimeOffset(1970, 1, 18, 0,
0, 0, TimeSpan.Zero),
                Rating = 3.6,
                Address = new Address()
```

```
                {
                    StreetAddress = "677 5th Ave",
                    City = "New York",
                    StateProvince = "NY",
                    PostalCode = "10022",
                    Country = "USA"
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "2",
                HotelName = "Twin Dome Motel",
                Description = "The hotel is situated in a  nineteenth
century plaza, which has been expanded and renovated to the highest
architectural standards to create a modern, functional and first-class
hotel in which art and unique historical elements coexist with the most
modern comforts.",
                DescriptionFr = "L'hôtel est situé dans une place du
XIXe siècle, qui a été agrandie et rénovée aux plus hautes normes
architecturales pour créer un hôtel moderne, fonctionnel et de première
classe dans lequel l'art et les éléments historiques uniques coexistent
avec le confort le plus moderne.",
                Category = "Boutique",
                Tags = new[] { "pool", "free wifi", "concierge" },
                ParkingIncluded = false,
                LastRenovationDate = new DateTimeOffset(1979, 2, 18, 0,
0, 0, TimeSpan.Zero),
                Rating = 3.60,
                Address = new Address()
                {
                    StreetAddress = "140 University Town Center Dr",
                    City = "Sarasota",
                    StateProvince = "FL",
                    PostalCode = "34243",
                    Country = "USA"
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "3",
                HotelName = "Triple Landscape Hotel",
                Description = "The Hotel stands out for its gastronomic
excellence under the management of William Dough, who advises on and
oversees all of the Hotel’s restaurant services.",
                DescriptionFr = "L'hôtel est situé dans une place du
XIXe siècle, qui a été agrandie et rénovée aux plus hautes normes
architecturales pour créer un hôtel moderne, fonctionnel et de première
classe dans lequel l'art et les éléments historiques uniques coexistent
avec le confort le plus moderne.",
                Category = "Resort and Spa",
                Tags = new[] { "air conditioning", "bar", "continental
breakfast" },
                ParkingIncluded = true,
```

```
                LastRenovationDate = new DateTimeOffset(2015, 9, 20, 0,
0, 0, TimeSpan.Zero),
                Rating = 4.80,
                Address = new Address()
                {
                    StreetAddress = "3393 Peachtree Rd",
                    City = "Atlanta",
                    StateProvince = "GA",
                    PostalCode = "30326",
                    Country = "USA"
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "4",
                HotelName = "Sublime Cliff Hotel",
                Description = "Sublime Cliff Hotel is located in the
heart of the historic center of Sublime in an extremely vibrant and
lively area within short walking distance to the sites and landmarks of
the city and is surrounded by the extraordinary beauty of churches,
buildings, shops and monuments. Sublime Cliff is part of a lovingly
restored 1800 palace.",
                DescriptionFr = "Le sublime Cliff Hotel est situé au
coeur du centre historique de sublime dans un quartier extrêmement
animé et vivant, à courte distance de marche des sites et monuments de
la ville et est entouré par l'extraordinaire beauté des églises, des
bâtiments, des commerces et Monuments. Sublime Cliff fait partie d'un
Palace 1800 restauré avec amour.",
                Category = "Boutique",
                Tags = new[] { "concierge", "view", "24-hour front desk
service" },
                ParkingIncluded = true,
                LastRenovationDate = new DateTimeOffset(1960, 2, 06, 0,
0, 0, TimeSpan.Zero),
                Rating = 4.60,
                Address = new Address()
                {
                    StreetAddress = "7400 San Pedro Ave",
                    City = "San Antonio",
                    StateProvince = "TX",
                    PostalCode = "78216",
                    Country = "USA"
                }
            })
        );

    try
    {
        IndexDocumentsResult result =
searchClient.IndexDocuments(batch);
    }
    catch (Exception)
    {
        // If for some reason any documents are dropped during
```

```
indexing, you can compensate by delaying and
        // retrying. This simple demo just logs the failed document
keys and continues.
        Console.WriteLine("Failed to index some of the documents:
{0}");
    }
}
```

Once you initialize the IndexDocumentsBatch object, you can send it to the index by calling IndexDocuments on your SearchClient object.

2. Add the following lines to Main(). Loading documents is done using SearchClient, but the operation also requires admin rights on the service, which is typically associated with SearchIndexClient. One way to set up this operation is to get SearchClient through SearchIndexClient (adminClient in this example).

```C#
SearchClient ingesterClient = adminClient.GetSearchClient(indexName);

// Load documents
Console.WriteLine("{0}", "Uploading documents...\n");
UploadDocuments(ingesterClient);
```

3. Because this is a console app that runs all commands sequentially, add a 2-second wait time between indexing and queries.

```C#
// Wait 2 seconds for indexing to complete before starting queries (for
demo and console-app purposes only)
Console.WriteLine("Waiting for indexing...\n");
System.Threading.Thread.Sleep(2000);
```

The 2-second delay compensates for indexing, which is asynchronous, so that all documents can be indexed before the queries are executed. Coding in a delay is typically only necessary in demos, tests, and sample applications.

3 - Search an index

You can get query results as soon as the first document is indexed, but actual testing of your index should wait until all documents are indexed.

This section adds two pieces of functionality: query logic, and results. For queries, use the Search method. This method takes search text (the query string) and other options.

The SearchResults class represents the results.

1. In **Program.cs**, create a **WriteDocuments** method that prints search results to the console.

```C#
// Write search results to console
private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.GetResults())
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}

private static void WriteDocuments(AutocompleteResults autoResults)
{
    foreach (AutocompleteItem result in autoResults.Results)
    {
        Console.WriteLine(result.Text);
    }

    Console.WriteLine();
}
```

2. Create a **RunQueries** method to execute queries and return results. Results are Hotel objects. This sample shows the method signature and the first query. This query demonstrates the Select parameter that lets you compose the result using selected fields from the document.

```C#
// Run queries, use WriteDocuments to print output
private static void RunQueries(SearchClient srchclient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    // Query 1
    Console.WriteLine("Query #1: Search on empty term '*' to return all
documents, showing a subset of fields...\n");

    options = new SearchOptions()
    {
        IncludeTotalCount = true,
        Filter = "",
        OrderBy = { "" }
    };
```

```
    options.Select.Add("HotelId");
    options.Select.Add("HotelName");
    options.Select.Add("Address/City");

    response = srchclient.Search<Hotel>("*", options);
    WriteDocuments(response);
```

3. In the second query, search on a term, add a filter that selects documents where Rating is greater than 4, and then sort by Rating in descending order. Filter is a boolean expression that is evaluated over IsFilterable fields in an index. Filter queries either include or exclude values. As such, there's no relevance score associated with a filter query.

```C#
// Query 2
Console.WriteLine("Query #2: Search on 'hotels', filter on 'Rating gt 4', sort by Rating in descending order...\n");

options = new SearchOptions()
{
    Filter = "Rating gt 4",
    OrderBy = { "Rating desc" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Rating");

response = srchclient.Search<Hotel>("hotels", options);
WriteDocuments(response);
```

4. The third query demonstrates searchFields, used to scope a full text search operation to specific fields.

```C#
// Query 3
Console.WriteLine("Query #3: Limit search to specific fields (pool in Tags field)...\n");

options = new SearchOptions()
{
    SearchFields = { "Tags" }
};

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Tags");
```

```
response = srchclient.Search<Hotel>("pool", options);
WriteDocuments(response);
```

5. The fourth query demonstrates facets, which can be used to structure a faceted navigation structure.

```C#
// Query 4
Console.WriteLine("Query #4: Facet on 'Category'...\n");

options = new SearchOptions()
{
    Filter = ""
};

options.Facets.Add("Category");

options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Category");

response = srchclient.Search<Hotel>("*", options);
WriteDocuments(response);
```

6. In the fifth query, return a specific document. A document lookup is a typical response to OnClick event in a result set.

```C#
// Query 5
Console.WriteLine("Query #5: Look up a specific document...\n");

Response<Hotel> lookupResponse;
lookupResponse = srchclient.GetDocument<Hotel>("3");

Console.WriteLine(lookupResponse.Value.HotelId);
```

7. The last query shows the syntax for autocomplete, simulating a partial user input of "sa" that resolves to two possible matches in the sourceFields associated with the suggester you defined in the index.

```C#
// Query 6
Console.WriteLine("Query #6: Call Autocomplete on HotelName that
starts with 'sa'...\n");
```

```
var autoresponse = srchclient.Autocomplete("sa", "sg");
WriteDocuments(autoresponse);
```

8. Add **RunQueries** to Main().

```C#
// Call the RunQueries method to invoke a series of queries
Console.WriteLine("Starting queries...\n");
RunQueries(srchclient);

// End the program
Console.WriteLine("{0}", "Complete. Press any key to end this
program...\n");
Console.ReadKey();
```

The previous queries show multiple ways of matching terms in a query: full-text search, filters, and autocomplete.

Full text search and filters are performed using the SearchClient.Search method. A search query can be passed in the `searchText` string, while a filter expression can be passed in the Filter property of the SearchOptions class. To filter without searching, just pass `"*"` for the `searchText` parameter of the Search method. To search without filtering, leave the `Filter` property unset, or don't pass in a `SearchOptions` instance at all.

Run the program

Press F5 to rebuild the app and run the program in its entirety.

Output includes messages from Console.WriteLine, with the addition of query information and results.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

In this C# quickstart, you worked through a set of tasks to create an index, load it with documents, and run queries. At different stages, we took shortcuts to simplify the code for readability and comprehension. Now that you're familiar with the basic concepts, try the next tutorial to call Cognitive Search APIs in the context of a web app.

Tutorial: Add search to web apps

Quickstart: Create an Azure Cognitive Search index in Java

Article • 02/17/2023 • 15 minutes to read

Create a Java console application that creates, loads, and queries a search index using Visual Studio Code, Java 11 SDK, and the Azure.Search.Documents client library in the Azure SDK for Java.. This article provides step-by-step instructions for creating the application. Alternatively, you can download and run the complete application.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

We used the following software and services to build and test this quickstart:

- Visual Studio Code
- Java extension for Visual Studio Code
- Java 11 SDK
- Create an Azure Cognitive Search service or find an existing service under your current subscription. You can use a free service for this quickstart.

Get a key and URL

Calls to the service require a URL endpoint and an access key on every request. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

Every request sent to your service requires an API key. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Set up your environment

Begin by opening Visual Studio Code and setting up a new project.

Create the project

1. Open Visual Studio Code.

2. Install the Extension Pack For Java.

3. Open the Command Palette **Ctrl+Shift+P**. Search for **Create Java Project**.

4. Select **Maven**.

5. Select **maven-archetype-quickstart**.

6. Select the latest version, currently **1.4**.

7. Enter **azure.search.sample** as the group ID.

8. Enter **azuresearchquickstart** as the artifact ID.

9. Select the folder to create the project in.

10. Finish project creation in the integrated terminal. Press enter to accept the default for "1.0-SNAPSHOT" and then type "y" to confirm the properties for your project.

```
Define value for property 'version' 1.0-SNAPSHOT: :
[INFO] Using property: package = main.java.app
Confirm properties configuration:
groupId: main.java.app
artifactId: demo
version: 1.0-SNAPSHOT
package: main.java.app
 Y: : y
```

11. Open the folder you created the project in.

Specify Maven dependencies

1. Open the pom.xml file and add the following dependencies

```XML
<dependencies>
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-search-documents</artifactId>
      <version>11.5.2</version>
    </dependency>
    <dependency>
      <groupId>com.azure</groupId>
      <artifactId>azure-core</artifactId>
      <version>1.34.0</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

2. Change the compiler Java version to 11

```XML
```

```
<maven.compiler.source>1.11</maven.compiler.source>
<maven.compiler.target>1.11</maven.compiler.target>
```

Create a search client

1. Open the `App` class under **src**, **main**, **java**, **azure**, **search**, **sample**. Add the following import directives

Java

```
import java.util.Arrays;
import java.util.ArrayList;
import java.time.OffsetDateTime;
import java.time.ZoneOffset;
import java.time.LocalDateTime;
import java.time.LocalDate;
import java.time.LocalTime;

import com.azure.core.credential.AzureKeyCredential;
import com.azure.core.util.Context;
import com.azure.search.documents.SearchClient;
import com.azure.search.documents.SearchClientBuilder;
import com.azure.search.documents.models.SearchOptions;
import com.azure.search.documents.indexes.SearchIndexClient;
import com.azure.search.documents.indexes.SearchIndexClientBuilder;
import com.azure.search.documents.indexes.models.IndexDocumentsBatch;
import com.azure.search.documents.indexes.models.SearchIndex;
import com.azure.search.documents.indexes.models.SearchSuggester;
import com.azure.search.documents.util.AutocompletePagedIterable;
import com.azure.search.documents.util.SearchPagedIterable;
```

2. The following example includes placeholders for a search service name, admin API key that grants create and delete permissions, and index name. Substitute valid values for all three placeholders. Create two clients: SearchIndexClient creates the index, and SearchClient loads and queries an existing index. Both need the service endpoint and an admin API key for authentication with create and delete rights.

Java

```
public static void main(String[] args) {
    var searchServiceEndpoint = "https://<your-
service>.search.windows.net";
    var adminKey = new AzureKeyCredential("<your-admin-key>");
    String indexName = "<index-name>";

    SearchIndexClient searchIndexClient = new
SearchIndexClientBuilder()
        .endpoint(searchServiceEndpoint)
```

```
        .credential(adminKey)
        .buildClient();

    SearchClient searchClient = new SearchClientBuilder()
        .endpoint(searchServiceEndpoint)
        .credential(adminKey)
        .indexName(indexName)
        .buildClient();
}
```

1 - Create an index

This quickstart builds a Hotels index that you'll load with hotel data and execute queries against. In this step, define the fields in the index. Each field definition includes a name, data type, and attributes that determine how the field is used.

In this example, synchronous methods of the azure-search-documents library are used for simplicity and readability. However, for production scenarios, you should use asynchronous methods to keep your app scalable and responsive. For example, you would use SearchAsyncClient instead of SearchClient.

1. Add an empty class definition to your project: **Hotel.java**

2. Copy the following code into **Hotel.java** to define the structure of a hotel document. Attributes on the field determine how it's used in an application. For example, the IsFilterable annotation must be assigned to every field that supports a filter expression

Java

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License.

package azure.search.sample;

import com.azure.search.documents.indexes.SearchableField;
import com.azure.search.documents.indexes.SimpleField;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.annotation.JsonInclude.Include;

import java.time.OffsetDateTime;

/**
 * Model class representing a hotel.
 */
@JsonInclude(Include.NON_NULL)
```

```java
public class Hotel {
    /**
     * Hotel ID
     */
    @JsonProperty("HotelId")
    @SimpleField(isKey = true)
    public String hotelId;

    /**
     * Hotel name
     */
    @JsonProperty("HotelName")
    @SearchableField(isSortable = true)
    public String hotelName;

    /**
     * Description
     */
    @JsonProperty("Description")
    @SearchableField(analyzerName = "en.microsoft")
    public String description;

    /**
     * French description
     */
    @JsonProperty("DescriptionFr")
    @SearchableField(analyzerName = "fr.lucene")
    public String descriptionFr;

    /**
     * Category
     */
    @JsonProperty("Category")
    @SearchableField(isFilterable = true, isSortable = true,
isFacetable = true)
    public String category;

    /**
     * Tags
     */
    @JsonProperty("Tags")
    @SearchableField(isFilterable = true, isFacetable = true)
    public String[] tags;

    /**
     * Whether parking is included
     */
    @JsonProperty("ParkingIncluded")
    @SimpleField(isFilterable = true, isSortable = true, isFacetable =
true)
    public Boolean parkingIncluded;

    /**
     * Last renovation time
     */
```

```
    @JsonProperty("LastRenovationDate")
    @SimpleField(isFilterable = true, isSortable = true, isFacetable =
true)
    public OffsetDateTime lastRenovationDate;

    /**
     * Rating
     */
    @JsonProperty("Rating")
    @SimpleField(isFilterable = true, isSortable = true, isFacetable =
true)
    public Double rating;

    /**
     * Address
     */
    @JsonProperty("Address")
    public Address address;

    @Override
    public String toString()
    {
        try
        {
            return new ObjectMapper().writeValueAsString(this);
        }
        catch (JsonProcessingException e)
        {
            e.printStackTrace();
            return "";
        }
    }
}
```

In the Azure.Search.Documents client library, you can use SearchableField and SimpleField to streamline field definitions.

- `SimpleField` can be any data type, is always non-searchable (it's ignored for full text search queries), and is retrievable (it's not hidden). Other attributes are off by default, but can be enabled. You might use a SimpleField for document IDs or fields used only in filters, facets, or scoring profiles. If so, be sure to apply any attributes that are necessary for the scenario, such as IsKey = true for a document ID.
- `SearchableField` must be a string, and is always searchable and retrievable. Other attributes are off by default, but can be enabled. Because this field type is searchable, it supports synonyms and the full complement of analyzer properties.

Whether you use the basic `SearchField` API or either one of the helper models, you must explicitly enable filter, facet, and sort attributes. For example, `isFilterable`,

`isSortable`, and `isFacetable` must be explicitly attributed, as shown in the sample above.

1. Add a second empty class definition to your project: **Address.cs**. Copy the following code into the class.

Java

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT License.

package azure.search.sample;

import com.azure.search.documents.indexes.SearchableField;
import com.fasterxml.jackson.annotation.JsonInclude;
import com.fasterxml.jackson.annotation.JsonProperty;
import com.fasterxml.jackson.annotation.JsonInclude.Include;

/**
 * Model class representing an address.
 */
@JsonInclude(Include.NON_NULL)
public class Address {
    /**
     * Street address
     */
    @JsonProperty("StreetAddress")
    @SearchableField
    public String streetAddress;

    /**
     * City
     */
    @JsonProperty("City")
    @SearchableField(isFilterable = true, isSortable = true,
isFacetable = true)
    public String city;

    /**
     * State or province
     */
    @JsonProperty("StateProvince")
    @SearchableField(isFilterable = true, isSortable = true,
isFacetable = true)
    public String stateProvince;

    /**
     * Postal code
     */
    @JsonProperty("PostalCode")
    @SearchableField(isFilterable = true, isSortable = true,
isFacetable = true)
    public String postalCode;
```

```
    /**
     * Country
     */
    @JsonProperty("Country")
    @SearchableField(isFilterable = true, isSortable = true,
isFacetable = true)
    public String country;
}
```

2. In **App.java**, create a `SearchIndex` object in the **main** method, and then call the `createOrUpdateIndex` method to create the index in your search service. The index also includes a `SearchSuggester` to enable autocomplete on the specified fields.

```
Java

// Create Search Index for Hotel model
searchIndexClient.createOrUpdateIndex(
    new SearchIndex(indexName,
SearchIndexClient.buildSearchFields(Hotel.class, null))
    .setSuggesters(new SearchSuggester("sg",
Arrays.asList("HotelName"))));
```

2 - Load Documents

Azure Cognitive Search searches over content stored in the service. In this step, you'll load JSON documents that conform to the hotel index you just created.

In Azure Cognitive Search, search documents are data structures that are both inputs to indexing and outputs from queries. As obtained from an external data source, document inputs might be rows in a database, blobs in Blob storage, or JSON documents on disk. In this example, we're taking a shortcut and embedding JSON documents for four hotels in the code itself.

When uploading documents, you must use an IndexDocumentsBatch object. An `IndexDocumentsBatch` object contains a collection of IndexActions, each of which contains a document and a property telling Azure Cognitive Search what action to perform (upload, merge, delete, and mergeOrUpload).

1. In **App.java**, create documents and index actions, and then pass them to `IndexDocumentsBatch`. The documents below conform to the hotels-quickstart index, as defined by the hotel class.

```
Java
```

```
// Upload documents in a single Upload request.
private static void uploadDocuments(SearchClient searchClient)
{
    var hotelList = new ArrayList<Hotel>();

    var hotel = new Hotel();
    hotel.hotelId = "1";
    hotel.hotelName = "Secret Point Motel";
    hotel.description = "The hotel is ideally located on the main
commercial artery of the city in the heart of New York. A few minutes
away is Time's Square and the historic centre of the city, as well as
other places of interest that make New York one of America's most
attractive and cosmopolitan cities.";
    hotel.descriptionFr = "L'hôtel est idéalement situé sur la
principale artère commerciale de la ville en plein cœur de New York. A
quelques minutes se trouve la place du temps et le centre historique de
la ville, ainsi que d'autres lieux d'intérêt qui font de New York l'une
des villes les plus attractives et cosmopolites de l'Amérique.";
    hotel.category = "Boutique";
    hotel.tags = new String[] { "pool", "air conditioning", "concierge"
};
    hotel.parkingIncluded = false;
    hotel.lastRenovationDate =
OffsetDateTime.of(LocalDateTime.of(LocalDate.of(1970, 1, 18),
LocalTime.of(0, 0)), ZoneOffset.UTC);
    hotel.rating = 3.6;
    hotel.address = new Address();
    hotel.address.streetAddress = "677 5th Ave";
    hotel.address.city = "New York";
    hotel.address.stateProvince = "NY";
    hotel.address.postalCode = "10022";
    hotel.address.country = "USA";
    hotelList.add(hotel);

    hotel = new Hotel();
    hotel.hotelId = "2";
    hotel.hotelName = "Twin Dome Motel";
    hotel.description = "The hotel is situated in a  nineteenth century
plaza, which has been expanded and renovated to the highest
architectural standards to create a modern, functional and first-class
hotel in which art and unique historical elements coexist with the most
modern comforts.";
    hotel.descriptionFr = "L'hôtel est situé dans une place du XIXe
siècle, qui a été agrandie et rénovée aux plus hautes normes
architecturales pour créer un hôtel moderne, fonctionnel et de première
classe dans lequel l'art et les éléments historiques uniques coexistent
avec le confort le plus moderne.";
    hotel.category = "Boutique";
    hotel.tags = new String[] { "pool", "free wifi", "concierge" };
    hotel.parkingIncluded = false;
    hotel.lastRenovationDate =
OffsetDateTime.of(LocalDateTime.of(LocalDate.of(1979, 2, 18),
LocalTime.of(0, 0)), ZoneOffset.UTC);
    hotel.rating = 3.60;
```

```
    hotel.address = new Address();
    hotel.address.streetAddress = "140 University Town Center Dr";
    hotel.address.city = "Sarasota";
    hotel.address.stateProvince = "FL";
    hotel.address.postalCode = "34243";
    hotel.address.country = "USA";
    hotelList.add(hotel);

    hotel = new Hotel();
    hotel.hotelId = "3";
    hotel.hotelName = "Triple Landscape Hotel";
    hotel.description = "The Hotel stands out for its gastronomic
excellence under the management of William Dough, who advises on and
oversees all of the Hotel’s restaurant services.";
    hotel.descriptionFr = "L'hôtel est situé dans une place du XIXe
siècle, qui a été agrandie et rénovée aux plus hautes normes
architecturales pour créer un hôtel moderne, fonctionnel et de première
classe dans lequel l'art et les éléments historiques uniques coexistent
avec le confort le plus moderne.";
    hotel.category = "Resort and Spa";
    hotel.tags = new String[] { "air conditioning", "bar", "continental
breakfast" };
    hotel.parkingIncluded = true;
    hotel.lastRenovationDate =
OffsetDateTime.of(LocalDateTime.of(LocalDate.of(2015, 9, 20),
LocalTime.of(0, 0)), ZoneOffset.UTC);
    hotel.rating = 4.80;
    hotel.address = new Address();
    hotel.address.streetAddress = "3393 Peachtree Rd";
    hotel.address.city = "Atlanta";
    hotel.address.stateProvince = "GA";
    hotel.address.postalCode = "30326";
    hotel.address.country = "USA";
    hotelList.add(hotel);

    hotel = new Hotel();
    hotel.hotelId = "4";
    hotel.hotelName = "Sublime Cliff Hotel";
    hotel.description = "Sublime Cliff Hotel is located in the heart of
the historic center of Sublime in an extremely vibrant and lively area
within short walking distance to the sites and landmarks of the city
and is surrounded by the extraordinary beauty of churches, buildings,
shops and monuments. Sublime Cliff is part of a lovingly restored 1800
palace.";
    hotel.descriptionFr = "Le sublime Cliff Hotel est situé au coeur du
centre historique de sublime dans un quartier extrêmement animé et
vivant, à courte distance de marche des sites et monuments de la ville
et est entouré par l'extraordinaire beauté des églises, des bâtiments,
des commerces et Monuments. Sublime Cliff fait partie d'un Palace 1800
restauré avec amour.";
    hotel.category = "Boutique";
    hotel.tags = new String[] { "concierge", "view", "24-hour front
desk service" };
    hotel.parkingIncluded = true;
    hotel.lastRenovationDate =
```

```
OffsetDateTime.of(LocalDateTime.of(LocalDate.of(1960, 2, 06),
LocalTime.of(0, 0)), ZoneOffset.UTC);
    hotel.rating = 4.60;
    hotel.address = new Address();
    hotel.address.streetAddress = "7400 San Pedro Ave";
    hotel.address.city = "San Antonio";
    hotel.address.stateProvince = "TX";
    hotel.address.postalCode = "78216";
    hotel.address.country = "USA";
    hotelList.add(hotel);

    var batch = new IndexDocumentsBatch<Hotel>();
    batch.addMergeOrUploadActions(hotelList);
    try
    {
        searchClient.indexDocuments(batch);
    }
    catch (Exception e)
    {
        e.printStackTrace();
        // If for some reason any documents are dropped during
indexing, you can compensate by delaying and
        // retrying. This simple demo just logs failure and continues
        System.err.println("Failed to index some of the documents");
    }
}
```

Once you initialize the `IndexDocumentsBatch` object, you can send it to the index by calling indexDocuments on your `SearchClient` object.

1. Add the following lines to `Main()`. Loading documents is done using `SearchClient`.

Java

```
// Upload sample hotel documents to the Search Index
uploadDocuments(searchClient);
```

2. Because this is a console app that runs all commands sequentially, add a 2-second wait time between indexing and queries.

Java

```
// Wait 2 seconds for indexing to complete before starting queries (for
demo and console-app purposes only)
System.out.println("Waiting for indexing...\n");
try
{
    Thread.sleep(2000);
}
catch (InterruptedException e)
```

```
{
}
```

The 2-second delay compensates for indexing, which is asynchronous, so that all documents can be indexed before the queries are executed. Coding in a delay is typically only necessary in demos, tests, and sample applications.

3 - Search an index

You can get query results as soon as the first document is indexed, but actual testing of your index should wait until all documents are indexed.

This section adds two pieces of functionality: query logic, and results. For queries, use the Search method. This method takes search text (the query string) and other options.

1. In **App.java**, create a WriteDocuments method that prints search results to the console.

   Java

   ```java
   // Write search results to console
   private static void WriteSearchResults(SearchPagedIterable
   searchResults)
   {
       searchResults.iterator().forEachRemaining(result ->
       {
           Hotel hotel = result.getDocument(Hotel.class);
           System.out.println(hotel);
       });

       System.out.println();
   }

   // Write autocomplete results to console
   private static void WriteAutocompleteResults(AutocompletePagedIterable
   autocompleteResults)
   {
       autocompleteResults.iterator().forEachRemaining(result ->
       {
           String text = result.getText();
           System.out.println(text);
       });

       System.out.println();
   }
   ```

2. Create a `RunQueries` method to execute queries and return results. Results are `Hotel` objects. This sample shows the method signature and the first query. This

query demonstrates the `Select` parameter that lets you compose the result using selected fields from the document.

Java

```
// Run queries, use WriteDocuments to print output
private static void RunQueries(SearchClient searchClient)
{
    // Query 1
    System.out.println("Query #1: Search on empty term '*' to return
all documents, showing a subset of fields...\n");

    SearchOptions options = new SearchOptions();
    options.setIncludeTotalCount(true);
    options.setFilter("");
    options.setOrderBy("");
    options.setSelect("HotelId", "HotelName", "Address/City");

    WriteSearchResults(searchClient.search("*", options,
Context.NONE));
}
```

3. In the second query, search on a term, add a filter that selects documents where Rating is greater than 4, and then sort by Rating in descending order. Filter is a boolean expression that is evaluated over `isFilterable` fields in an index. Filter queries either include or exclude values. As such, there's no relevance score associated with a filter query.

Java

```
// Query 2
System.out.println("Query #2: Search on 'hotels', filter on 'Rating gt
4', sort by Rating in descending order...\n");

options = new SearchOptions();
options.setFilter("Rating gt 4");
options.setOrderBy("Rating desc");
options.setSelect("HotelId", "HotelName", "Rating");

WriteSearchResults(searchClient.search("hotels", options,
Context.NONE));
```

4. The third query demonstrates `searchFields`, used to scope a full text search operation to specific fields.

Java

```
// Query 3
System.out.println("Query #3: Limit search to specific fields (pool in
Tags field)...\n");

options = new SearchOptions();
options.setSearchFields("Tags");

options.setSelect("HotelId", "HotelName", "Tags");

WriteSearchResults(searchClient.search("pool", options, Context.NONE));
```

5. The fourth query demonstrates facets, which can be used to structure a faceted navigation structure.

Java

```
// Query 4
System.out.println("Query #4: Facet on 'Category'...\n");

options = new SearchOptions();
options.setFilter("");
options.setFacets("Category");
options.setSelect("HotelId", "HotelName", "Category");

WriteSearchResults(searchClient.search("*", options, Context.NONE));
```

6. In the fifth query, return a specific document.

Java

```
// Query 5
System.out.println("Query #5: Look up a specific document...\n");

Hotel lookupResponse = searchClient.getDocument("3", Hotel.class);
System.out.println(lookupResponse.hotelId);
System.out.println();
```

7. The last query shows the syntax for autocomplete, simulating a partial user input of "s" that resolves to two possible matches in the `sourceFields` associated with the suggester you defined in the index.

Java

```
// Query 6
System.out.println("Query #6: Call Autocomplete on HotelName that
starts with 's'...\n");

WriteAutocompleteResults(searchClient.autocomplete("s", "sg"));
```

8. Add RunQueries to Main().

```Java
// Call the RunQueries method to invoke a series of queries
System.out.println("Starting queries...\n");
RunQueries(searchClient);

// End the program
System.out.println("Complete.\n");
```

The previous queries show multiple ways of matching terms in a query: full-text search, filters, and autocomplete.

Full text search and filters are performed using the SearchClient.search method. A search query can be passed in the `searchText` string, while a filter expression can be passed in the `filter` property of the SearchOptions class. To filter without searching, just pass "*" for the `searchText` parameter of the `search` method. To search without filtering, leave the `filter` property unset, or don't pass in a `SearchOptions` instance at all.

Run the program

Press F5 to rebuild the app and run the program in its entirety.

Output includes messages from System.out.println, with the addition of query information and results.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next Steps

In this Java quickstart, you worked through a set of tasks to create an index, load it with documents, and run queries. At different stages, we took shortcuts to simplify the code for readability and comprehension. Now that you're familiar with the basic concepts, try the next tutorial to call Cognitive Search APIs in the context of a web app.

Tutorial: Add search to web apps

Additional resources

Documentation

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Azure Cognitive Search client library for Java

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Create Index (Azure Cognitive Search REST API)

Define an index schema for fields and other constructs in an Azure Cognitive Search index.

Quickstart: Create a search index using REST APIs - Azure Cognitive Search

In this REST API quickstart, learn how to call the Azure Cognitive Search REST APIs using Postman.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Show 5 more

Quickstart: Create an Azure Cognitive Search index using the JavaScript SDK

Article • 01/27/2023 • 14 minutes to read

Use the JavaScript/TypeScript SDK for Azure Cognitive Search to create a Node.js application in JavaScript that creates, loads, and queries a search index.

This article demonstrates how to create the application step by step. Alternatively, you can download the source code and data and run the application from the command line.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

Before you begin, have the following tools and services:

- Visual Studio Code or another IDE
- Node.js and npm
- Azure Cognitive Search. Create a service or find an existing service. You can use a free service for this quickstart.

Set up your project

Start by getting the endpoint and key for your search service. Then create a new project with npm as outlined below.

Copy a key and endpoint

Calls to the service require a URL endpoint and an access key on every request. As a first step, find the API key and URL to add to your project. You'll specify both values when creating the client in a later step.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service, required if you're creating or deleting objects. There are two interchangeable primary and secondary keys. You can use either one.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Create a new npm project

Begin by opening VS Code and its integrated terminal or another terminal such as the Node.js command prompt.

1. Create a development directory, giving it the name `quickstart` :

Windows Command Prompt

```
mkdir quickstart
cd quickstart
```

2. Initialize an empty project with npm by running the following command. To fully initialize the project, press Enter multiple times to accept the default values, except for the License, which you should set to "MIT".

Windows Command Prompt

```
npm init
```

3. Install `@azure/search-documents`, the JavaScript/TypeScript SDK for Azure Cognitive Search.

Windows Command Prompt

```
npm install @azure/search-documents
```

4. Install `dotenv`, which is used to import the environment variables such as your search service name and API key.

Windows Command Prompt

```
npm install dotenv
```

5. Confirm that you've configured the projects and its dependencies by checking that your **package.json** file looks similar to the following json:

JSON

```
{
  "name": "quickstart",
  "version": "1.0.0",
  "description": "Azure Cognitive Search Quickstart",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "Azure",
    "Search"
  ],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "@azure/search-documents": "^11.3.0",
    "dotenv": "^16.0.2"
  }
}
```

6. Create a file **.env** to hold your search service parameters:

```
SEARCH_API_KEY=<search-admin-key>
SEARCH_API_ENDPOINT=https://<search-service-name>.search.windows.net
```

Replace the `<search-service-name>` value with the name of your search service. Replace `<search-admin-key>` with the key value you recorded earlier.

Create index.js file

Next we create an **index.js** file, which is the main file that will host our code.

At the top of this file, we import the `@azure/search-documents` library:

JavaScript

```
const { SearchIndexClient, SearchClient, AzureKeyCredential, odata } =
require("@azure/search-documents");
```

Next, we need to require the `dotenv` package to read in the parameters from the **.env** file as follows:

JavaScript

```
// Load the .env file if it exists
require("dotenv").config();

// Getting endpoint and apiKey from .env file
const endpoint = process.env.SEARCH_API_ENDPOINT || "";
const apiKey = process.env.SEARCH_API_KEY || "";
```

With our imports and environment variables in place, we're ready to define the main function.

Most of the functionality in the SDK is asynchronous so we make our main function `async`. We also include a `main().catch()` below the main function to catch and log any errors encountered:

JavaScript

```
async function main() {
    console.log(`Running Azure Cognitive Search JavaScript quickstart...`);
    if (!endpoint || !apiKey) {
        console.log("Make sure to set valid values for endpoint and apiKey
with proper authorization.");
        return;
    }

    // remaining quickstart code will go here
}

main().catch((err) => {
    console.error("The sample encountered an error:", err);
});
```

With that in place, we're ready to create an index.

1 - Create index

Create a file **hotels_quickstart_index.json**. This file defines how Azure Cognitive Search works with the documents you'll be loading in the next step. Each field will be identified by a `name` and have a specified `type`. Each field also has a series of index attributes that specify whether Azure Cognitive Search can search, filter, sort, and facet upon the field. Most of the fields are simple data types, but some, like `AddressType` are complex types that allow you to create rich data structures in your index. You can read more about supported data types and index attributes described in Create Index (REST).

Add the following content to **hotels_quickstart_index.json** or download the file.

JSON

```
{
    "name": "hotels-quickstart",
    "fields": [
        {
            "name": "HotelId",
            "type": "Edm.String",
            "key": true,
            "filterable": true
        },
        {
            "name": "HotelName",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "sortable": true,
            "facetable": false
        },
        {
            "name": "Description",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "sortable": false,
            "facetable": false,
            "analyzerName": "en.lucene"
        },
        {
            "name": "Description_fr",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "sortable": false,
            "facetable": false,
            "analyzerName": "fr.lucene"
```

```
        },
        {
            "name": "Category",
            "type": "Edm.String",
            "searchable": true,
            "filterable": true,
            "sortable": true,
            "facetable": true
        },
        {
            "name": "Tags",
            "type": "Collection(Edm.String)",
            "searchable": true,
            "filterable": true,
            "sortable": false,
            "facetable": true
        },
        {
            "name": "ParkingIncluded",
            "type": "Edm.Boolean",
            "filterable": true,
            "sortable": true,
            "facetable": true
        },
        {
            "name": "LastRenovationDate",
            "type": "Edm.DateTimeOffset",
            "filterable": true,
            "sortable": true,
            "facetable": true
        },
        {
            "name": "Rating",
            "type": "Edm.Double",
            "filterable": true,
            "sortable": true,
            "facetable": true
        },
        {
            "name": "Address",
            "type": "Edm.ComplexType",
            "fields": [
                {
                    "name": "StreetAddress",
                    "type": "Edm.String",
                    "filterable": false,
                    "sortable": false,
                    "facetable": false,
                    "searchable": true
                },
                {
                    "name": "City",
                    "type": "Edm.String",
                    "searchable": true,
                    "filterable": true,
```

```
                    "sortable": true,
                    "facetable": true
                },
                {
                    "name": "StateProvince",
                    "type": "Edm.String",
                    "searchable": true,
                    "filterable": true,
                    "sortable": true,
                    "facetable": true
                },
                {
                    "name": "PostalCode",
                    "type": "Edm.String",
                    "searchable": true,
                    "filterable": true,
                    "sortable": true,
                    "facetable": true
                },
                {
                    "name": "Country",
                    "type": "Edm.String",
                    "searchable": true,
                    "filterable": true,
                    "sortable": true,
                    "facetable": true
                }
            ]
        }
    ],
    "suggesters": [
        {
            "name": "sg",
            "searchMode": "analyzingInfixMatching",
            "sourceFields": [
                "HotelName"
            ]
        }
    ]
}
```

With our index definition in place, we want to import **hotels_quickstart_index.json** at the top of **index.js** so the main function can access the index definition.

JavaScript

```
const indexDefinition = require('./hotels_quickstart_index.json');
```

Within the main function, we then create a `SearchIndexClient`, which is used to create and manage indexes for Azure Cognitive Search.

JavaScript

```
const indexClient = new SearchIndexClient(endpoint, new
AzureKeyCredential(apiKey));
```

Next, we want to delete the index if it already exists. This operation is a common practice for test/demo code.

We do this by defining a simple function that tries to delete the index.

JavaScript

```
async function deleteIndexIfExists(indexClient, indexName) {
    try {
        await indexClient.deleteIndex(indexName);
        console.log('Deleting index...');
    } catch {
        console.log('Index does not exist yet.');
    }
}
```

To run the function, we extract the index name from the index definition and pass the `indexName` along with the `indexClient` to the `deleteIndexIfExists()` function.

JavaScript

```
const indexName = indexDefinition["name"];

console.log('Checking if index exists...');
await deleteIndexIfExists(indexClient, indexName);
```

After that, we're ready to create the index with the `createIndex()` method.

JavaScript

```
console.log('Creating index...');
let index = await indexClient.createIndex(indexDefinition);

console.log(`Index named ${index.name} has been created.`);
```

Run the sample

At this point, you're ready to run the sample. Use a terminal window to run the following command:

Windows Command Prompt

```
node index.js
```

If you downloaded the source code and haven't installed the required packages yet, run `npm install` first.

You should see a series of messages describing the actions being taken by the program.

Open the **Overview** of your search service in the Azure portal. Select the **Indexes** tab. You should see something like the following example:

In the next step, you'll add data to index.

2 - Load documents

In Azure Cognitive Search, documents are data structures that are both inputs to indexing and outputs from queries. You can push such data to the index or use an indexer. In this case, we'll programatically push the documents to the index.

Document inputs might be rows in a database, blobs in Blob storage, or, as in this sample, JSON documents on disk. You can either download hotels.json or create your own **hotels.json** file with the following content:

JSON

```
{
    "value": [
        {
            "HotelId": "1",
            "HotelName": "Secret Point Motel",
            "Description": "The hotel is ideally located on the main
commercial artery of the city in the heart of New York. A few minutes away
is Time's Square and the historic centre of the city, as well as other
```

```
places of interest that make New York one of America's most attractive and
cosmopolitan cities.",
            "Description_fr": "L'hôtel est idéalement situé sur la
principale artère commerciale de la ville en plein cœur de New York. A
quelques minutes se trouve la place du temps et le centre historique de la
ville, ainsi que d'autres lieux d'intérêt qui font de New York l'une des
villes les plus attractives et cosmopolites de l'Amérique.",
            "Category": "Boutique",
            "Tags": ["pool", "air conditioning", "concierge"],
            "ParkingIncluded": false,
            "LastRenovationDate": "1970-01-18T00:00:00Z",
            "Rating": 3.6,
            "Address": {
                "StreetAddress": "677 5th Ave",
                "City": "New York",
                "StateProvince": "NY",
                "PostalCode": "10022"
            }
        },
        {
            "HotelId": "2",
            "HotelName": "Twin Dome Motel",
            "Description": "The hotel is situated in a  nineteenth century
plaza, which has been expanded and renovated to the highest architectural
standards to create a modern, functional and first-class hotel in which art
and unique historical elements coexist with the most modern comforts.",
            "Description_fr": "L'hôtel est situé dans une place du XIXe
siècle, qui a été agrandie et rénovée aux plus hautes normes architecturales
pour créer un hôtel moderne, fonctionnel et de première classe dans lequel
l'art et les éléments historiques uniques coexistent avec le confort le plus
moderne.",
            "Category": "Boutique",
            "Tags": ["pool", "free wifi", "concierge"],
            "ParkingIncluded": "false",
            "LastRenovationDate": "1979-02-18T00:00:00Z",
            "Rating": 3.6,
            "Address": {
                "StreetAddress": "140 University Town Center Dr",
                "City": "Sarasota",
                "StateProvince": "FL",
                "PostalCode": "34243"
            }
        },
        {
            "HotelId": "3",
            "HotelName": "Triple Landscape Hotel",
            "Description": "The Hotel stands out for its gastronomic
excellence under the management of William Dough, who advises on and
oversees all of the Hotel’s restaurant services.",
            "Description_fr": "L'hôtel est situé dans une place du XIXe
siècle, qui a été agrandie et rénovée aux plus hautes normes architecturales
pour créer un hôtel moderne, fonctionnel et de première classe dans lequel
l'art et les éléments historiques uniques coexistent avec le confort le plus
moderne.",
            "Category": "Resort and Spa",
```

```
            "Tags": ["air conditioning", "bar", "continental breakfast"],
            "ParkingIncluded": "true",
            "LastRenovationDate": "2015-09-20T00:00:00Z",
            "Rating": 4.8,
            "Address": {
                "StreetAddress": "3393 Peachtree Rd",
                "City": "Atlanta",
                "StateProvince": "GA",
                "PostalCode": "30326"
            }
        },
        {
            "HotelId": "4",
            "HotelName": "Sublime Cliff Hotel",
            "Description": "Sublime Cliff Hotel is located in the heart of
the historic center of Sublime in an extremely vibrant and lively area
within short walking distance to the sites and landmarks of the city and is
surrounded by the extraordinary beauty of churches, buildings, shops and
monuments. Sublime Cliff is part of a lovingly restored 1800 palace.",
            "Description_fr": "Le sublime Cliff Hotel est situé au coeur du
centre historique de sublime dans un quartier extrêmement animé et vivant, à
courte distance de marche des sites et monuments de la ville et est entouré
par l'extraordinaire beauté des églises, des bâtiments, des commerces et
Monuments. Sublime Cliff fait partie d'un Palace 1800 restauré avec amour.",
            "Category": "Boutique",
            "Tags": ["concierge", "view", "24-hour front desk service"],
            "ParkingIncluded": true,
            "LastRenovationDate": "1960-02-06T00:00:00Z",
            "Rating": 4.6,
            "Address": {
                "StreetAddress": "7400 San Pedro Ave",
                "City": "San Antonio",
                "StateProvince": "TX",
                "PostalCode": "78216"
            }
        }
    ]
}
```

Similar to what we did with the indexDefinition, we also need to import `hotels.json` at the top of **index.js** so that the data can be accessed in our main function.

JavaScript

```javascript
const hotelData = require('./hotels.json');
```

To index data into the search index, we now need to create a `SearchClient`. While the `SearchIndexClient` is used to create and manage an index, the `SearchClient` is used to upload documents and query the index.

There are two ways to create a `SearchClient`. The first option is to create a `SearchClient` from scratch:

JavaScript

```
 const searchClient = new SearchClient(endpoint, indexName, new
AzureKeyCredential(apiKey));
```

Alternatively, you can use the `getSearchClient()` method of the `SearchIndexClient` to create the `SearchClient`:

JavaScript

```
const searchClient = indexClient.getSearchClient(indexName);
```

Now that the client is defined, upload the documents into the search index. In this case, we use the `mergeOrUploadDocuments()` method, which will upload the documents or merge them with an existing document if a document with the same key already exists.

JavaScript

```
console.log('Uploading documents...');
let indexDocumentsResult = await
searchClient.mergeOrUploadDocuments(hotelData['value']);

console.log(`Index operations succeeded:
${JSON.stringify(indexDocumentsResult.results[0].succeeded)}`);
```

Run the program again with `node index.js`. You should see a slightly different set of messages from those you saw in Step 1. This time, the index *does* exist, and you should see a message about deleting it before the app creates the new index and posts data to it.

Before we run the queries in the next step, define a function to have the program wait for one second. This is done just for test/demo purposes to ensure the indexing finishes and that the documents are available in the index for our queries.

JavaScript

```
function sleep(ms) {
    var d = new Date();
    var d2 = null;
    do {
        d2 = new Date();
```

```
    } while (d2 - d < ms);
}
```

To have the program wait for one second, call the `sleep` function like below:

JavaScript

```
sleep(1000);
```

3 - Search an index

With an index created and documents uploaded, you're ready to send queries to the index. In this section, we'll send five different queries to the search index to demonstrate different pieces of query functionality available to you.

The queries are written in a `sendQueries()` function that we'll call in the main function as follows:

JavaScript

```
await sendQueries(searchClient);
```

Queries are sent using the `search()` method of `searchClient`. The first parameter is the search text and the second parameter specifies search options.

The first query searches `*`, which is equivalent to searching everything and selects three of the fields in the index. It's a best practice to only `select` the fields you need because pulling back unnecessary data can add latency to your queries.

The `searchOptions` for this query also has `includeTotalCount` set to `true`, which will return the number of matching results found.

JavaScript

```
async function sendQueries(searchClient) {
    console.log('Query #1 - search everything:');
    let searchOptions = {
        includeTotalCount: true,
        select: ["HotelId", "HotelName", "Rating"]
    };

    let searchResults = await searchClient.search("*", searchOptions);
    for await (const result of searchResults.results) {
        console.log(`${JSON.stringify(result.document)}`);
    }
```

```
    console.log(`Result count: ${searchResults.count}`);

    // remaining queries go here
}
```

The remaining queries outlined below should also be added to the `sendQueries()` function. They're separated here for readability.

In the next query, we specify the search term `"wifi"` and also include a filter to only return results where the state is equal to `'FL'`. Results are also ordered by the Hotel's `Rating`.

JavaScript

```
console.log('Query #2 - Search with filter, orderBy, and select:');
let state = 'FL';
searchOptions = {
    filter: odata`Address/StateProvince eq ${state}`,
    orderBy: ["Rating desc"],
    select: ["HotelId", "HotelName", "Rating"]
};

searchResults = await searchClient.search("wifi", searchOptions);
for await (const result of searchResults.results) {
    console.log(`${JSON.stringify(result.document)}`);
}
```

Next, the search is limited to a single searchable field using the `searchFields` parameter. This approach is a great option to make your query more efficient if you know you're only interested in matches in certain fields.

JavaScript

```
console.log('Query #3 - Limit searchFields:');
searchOptions = {
    select: ["HotelId", "HotelName", "Rating"],
    searchFields: ["HotelName"]
};

searchResults = await searchClient.search("sublime cliff", searchOptions);
for await (const result of searchResults.results) {
    console.log(`${JSON.stringify(result.document)}`);
}
console.log();
```

Another common option to include in a query is `facets`. Facets allow you to build out filters on your UI to make it easy for users to know what values they can filter down to.

JavaScript

```
console.log('Query #4 - Use facets:');
searchOptions = {
    facets: ["Category"],
    select: ["HotelId", "HotelName", "Rating"],
    searchFields: ["HotelName"]
};

searchResults = await searchClient.search("*", searchOptions);
for await (const result of searchResults.results) {
    console.log(`${JSON.stringify(result.document)}`);
}
```

The final query uses the `getDocument()` method of the `searchClient`. This allows you to efficiently retrieve a document by its key.

JavaScript

```
console.log('Query #5 - Lookup document:');
let documentResult = await searchClient.getDocument(key='3')
console.log(`HotelId: ${documentResult.HotelId}; HotelName:
${documentResult.HotelName}`)
```

Run the sample

Run the program with `node index.js`. Now, in addition to the previous steps, the queries will be sent and the results written to the console.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember the limit of three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

In this JavaScript quickstart, you worked through a series of tasks to create an index, load it with documents, and run queries. To continue learning, try the following tutorial and sample as a springboard for trying suggesters (type-ahead or autocomplete queries), filters, and faceted navigation.

- Tutorial: Add search to web apps
- Sample: Build a React front-end for Azure Cognitive Search

Quickstart: Create an Azure Cognitive Search index using REST APIs

Article • 03/24/2023 • 9 minutes to read

This article explains how to formulate requests interactively using the Azure Cognitive Search REST APIs and a REST client for sending and receiving requests.

The article uses the Postman app. You can download and import a Postman collection if you prefer to use predefined requests.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- Postman app, used for sending requests to Azure Cognitive Search.
- Create an Azure Cognitive Search service or find an existing service under your current subscription. You can use a free service for this quickstart.

Copy a key and URL

REST calls require the service URL and an access key on every request. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Connect to Azure Cognitive Search

Connection information is specified in the URI endpoint. Collection variables are used to represent the search service name and API keys. A typical URI in this quickstart looks like this:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-quickstart?api-version=2020-06-30
```

Notice the HTTPS prefix, the name of the service (variable, the name of an object (in this case, the name of an index in the indexes collection), and the api-version. The api-version is a required.

Request header composition includes two elements: `Content-Type` and the `api-key` used to authenticate to Azure Cognitive Search. The `api-key` is specified as variable, and it's also required.

For the requests to succeed, you'll need to provide the service name and api-key as collection variables.

1. Open the Postman app and import the collection.

2. Select the collection's access menu, select **Edit**, and provide the service name and key of your search service.

1 - Create an index

In Azure Cognitive Search, you usually create the index before loading it with data. The Create Index REST API is used for this task.

The URL is extended to include the `hotels-quickstart` index name.

1. Set the verb to **PUT**.
2. Copy in this URL `https://{{service-name}}.search.windows.net/indexes/hotels-quickstart?api-version=2020-06-30`.
3. Provide the index definition (copy-ready code is provided next) in the body of the request.
4. Select **Send**.

```
{
    "name": "hotels-quickstart",
    "fields": [
        {"name": "HotelId", "type": "Edm.String", "key": true, "filterable": true},
        {"name": "HotelName", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": true, "facetable": false},
        {"name": "Description", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false, "analyzer": "en.lucene"},
        {"name": "Category", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "Tags", "type": "Collection(Edm.String)", "searchable": true, "filterable": true, "sortable": false, "facetable": true},
        {"name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true, "sortable": true, "facetable": true},
        {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset", "filterable": true, "sortable": true, "facetable": true},
        {"name": "Rating", "type": "Edm.Double", "filterable": true, "sortable": true, "facetable": true},
        {"name": "Address", "type": "Edm.ComplexType",
        "fields": [
        {"name": "StreetAddress", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false, "searchable": true},
        {"name": "City", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "StateProvince", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "PostalCode", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "Country", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true}
        ]
    }
  ]
}
```

Index definition

The fields collection defines document structure. Each document must have these fields, and each field must have a data type. String fields are used in full text search. If you need numeric data to be searchable, you'll need to cast numeric data as strings.

Attributes on the field determine allowed action. The REST APIs allow many actions by default. For example, all strings are searchable, retrievable, filterable, and facetable by default. Often, you only have to set attributes when you need to turn off a behavior.

JSON

```
{
    "name": "hotels-quickstart",
    "fields": [
        {"name": "HotelId", "type": "Edm.String", "key": true, "filterable":
true},
        {"name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": true, "facetable": false},
        {"name": "Description", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"en.lucene"},
        {"name": "Category", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "Tags", "type": "Collection(Edm.String)", "searchable":
true, "filterable": true, "sortable": false, "facetable": true},
        {"name": "ParkingIncluded", "type": "Edm.Boolean", "filterable":
true, "sortable": true, "facetable": true},
        {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset",
"filterable": true, "sortable": true, "facetable": true},
        {"name": "Rating", "type": "Edm.Double", "filterable": true,
"sortable": true, "facetable": true},
        {"name": "Address", "type": "Edm.ComplexType",
        "fields": [
        {"name": "StreetAddress", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false, "searchable": true},
        {"name": "City", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "StateProvince", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "PostalCode", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "Country", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true}
        ]
     }
  ]
}
```

When you submit this request, you should get an HTTP 201 response, indicating the index was created successfully. You can verify this action in the portal, but note that the portal page has refresh intervals so it could take a minute or two to catch up.

> **Tip**
>
> If you get HTTP 504, verify the URL specifies HTTPS. If you see HTTP 400 or 404, check the request body to verify there were no copy-paste errors. An HTTP 403 typically indicates a problem with the api-key (either an invalid key or a syntax problem with how the api-key is specified).

2 - Load documents

Creating the index and populating the index are separate steps. In Azure Cognitive Search, the index contains all searchable data. In this scenario, the data is provided as JSON documents. The Add, Update, or Delete Documents REST API is used for this task.

The URL is extended to include the `docs` collections and `index` operation.

1. Set the verb to **POST**.
2. Copy in this URL `https://{{service-name}}.search.windows.net/indexes/hotels-quickstart/docs/index?api-version=2020-06-30`.
3. Provide the JSON documents (copy-ready code is next) in the body of the request.
4. Select **Send**.

JSON documents to load into the index

The Request Body contains four documents to be added to the hotels index.

JSON

```
{
    "value": [
```

```
    {
    "@search.action": "upload",
    "HotelId": "1",
    "HotelName": "Secret Point Motel",
    "Description": "The hotel is ideally located on the main commercial
artery of the city in the heart of New York. A few minutes away is Time's
Square and the historic centre of the city, as well as other places of
interest that make New York one of America's most attractive and
cosmopolitan cities.",
    "Category": "Boutique",
    "Tags": [ "pool", "air conditioning", "concierge" ],
    "ParkingIncluded": false,
    "LastRenovationDate": "1970-01-18T00:00:00Z",
    "Rating": 3.60,
    "Address":
        {
        "StreetAddress": "677 5th Ave",
        "City": "New York",
        "StateProvince": "NY",
        "PostalCode": "10022",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "2",
    "HotelName": "Twin Dome Motel",
    "Description": "The hotel is situated in a  nineteenth century plaza,
which has been expanded and renovated to the highest architectural standards
to create a modern, functional and first-class hotel in which art and unique
historical elements coexist with the most modern comforts.",
    "Category": "Boutique",
    "Tags": [ "pool", "free wifi", "concierge" ],
    "ParkingIncluded": false,
    "LastRenovationDate": "1979-02-18T00:00:00Z",
    "Rating": 3.60,
    "Address":
        {
        "StreetAddress": "140 University Town Center Dr",
        "City": "Sarasota",
        "StateProvince": "FL",
        "PostalCode": "34243",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "3",
    "HotelName": "Triple Landscape Hotel",
    "Description": "The Hotel stands out for its gastronomic excellence
under the management of William Dough, who advises on and oversees all of
the Hotel’s restaurant services.",
    "Category": "Resort and Spa",
    "Tags": [ "air conditioning", "bar", "continental breakfast" ],
    "ParkingIncluded": true,
```

```
    "LastRenovationDate": "2015-09-20T00:00:00Z",
    "Rating": 4.80,
    "Address":
        {
        "StreetAddress": "3393 Peachtree Rd",
        "City": "Atlanta",
        "StateProvince": "GA",
        "PostalCode": "30326",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "4",
    "HotelName": "Sublime Cliff Hotel",
    "Description": "Sublime Cliff Hotel is located in the heart of the
historic center of Sublime in an extremely vibrant and lively area within
short walking distance to the sites and landmarks of the city and is
surrounded by the extraordinary beauty of churches, buildings, shops and
monuments. Sublime Cliff is part of a lovingly restored 1800 palace.",
    "Category": "Boutique",
    "Tags": [ "concierge", "view", "24-hour front desk service" ],
    "ParkingIncluded": true,
    "LastRenovationDate": "1960-02-06T00:00:00Z",
    "Rating": 4.60,
    "Address":
        {
        "StreetAddress": "7400 San Pedro Ave",
        "City": "San Antonio",
        "StateProvince": "TX",
        "PostalCode": "78216",
        "Country": "USA"
        }
    }
  ]
}
```

In a few seconds, you should see an HTTP 201 response in the session list. This indicates the documents were created successfully.

If you get a 207, at least one document failed to upload. If you get a 404, you have a syntax error in either the header or body of the request: verify you changed the endpoint to include `/docs/index`.

> **Tip**
>
> For selected data sources, you can choose the alternative *indexer* approach which simplifies and reduces the amount of code required for indexing. For more information, see **Indexer operations**.

3 - Search an index

Now that an index and document set are loaded, you can issue queries against them using Search Documents REST API.

The URL is extended to include a query expression, specified using the search operator.

1. Set the verb to **GET**.
2. Copy in this URL `https://{{service-name}}.search.windows.net/indexes/hotels-quickstart/docs?search=*&$count=true&api-version=2020-06-30`.
3. Select **Send**.

This query is an empty and returns a count of the documents in the search results. The request and response should look similar to the following screenshot for Postman after you select **Send**. The status code should be 200.

Try a few other query examples to get a feel for the syntax. You can do a string search, verbatim $filter queries, limit the results set, scope the search to specific fields, and more.

```
# Query example 1 - Search on restaurant and wifi
# Return only the HotelName, Description, and Tags fields
https://{{service-name}}.search.windows.net/indexes/hotels-quickstart/docs?
search=restaurant wifi&$count=true&$select=HotelName,Description,Tags&api-
version=2020-06-30

# Query example 2 - Apply a filter to the index to find hotels rated 4 or
highter
# Returns the HotelName and Rating. Two documents match
https://{service-name}}.search.windows.net/indexes/hotels-quickstart/docs?
search=*&$filter=Rating gt 4&$select=HotelName,Rating&api-version=2020-06-30

# Query example 3 - Take the top two results, and show only HotelName and
Category in the results
https://{service-name}}.search.windows.net/indexes/hotels-quickstart/docs?
search=boutique&$top=2&$select=HotelName,Category&api-version=2020-06-30

# Query example 4 - Sort by a specific field (Address/City) in ascending
order
https://{service-name}}.search.windows.net/indexes/hotels-quickstart/docs?
```

```
search=pool&$orderby=Address/City asc&$select=HotelName, Address/City, Tags,
Rating&api-version=2020-06-30
```

Get index properties

You can also use Get Statistics to query for document counts and index size:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-quickstart/stats?
api-version=2020-06-30
```

Adding `/stats` to your URL returns index information. In Postman, your request should look similar to the following, and the response includes a document count and space used in bytes.

Notice that the api-version syntax differs. For this request, use `?` to append the api-version. The `?` separates the URL path from the query string, while & separates each 'name=value' pair in the query string. For this query, api-version is the first and only item in the query string.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

Now that you know how to perform core tasks, you can move forward with more REST API calls for advanced features, such as indexers or setting up an enrichment pipeline that adds content transformations to indexing. For your next step, we recommend the following link:

Tutorial: Use REST and AI to generate searchable content from Azure blobs

Quickstart: Create an Azure Cognitive Search index in PowerShell using REST APIs

Article • 03/10/2023 • 10 minutes to read

This article walks you through the process of creating, loading, and querying an Azure Cognitive Search index using PowerShell and the Azure Cognitive Search REST APIs. This article explains how to run PowerShell commands interactively. Alternatively, you can download and run a PowerShell script that performs the same operations.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

The following services and tools are required for this quickstart.

- PowerShell 5.1 or later, using Invoke-RestMethod for sequential and interactive steps.
- Create an Azure Cognitive Search service or find an existing service under your current subscription. You can use a free service for this quickstart.

Copy a key and URL

REST calls require the service URL and an access key on every request. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Connect to Azure Cognitive Search

1. In PowerShell, create a **$headers** object to store the content-type and API key. Replace the admin API key (YOUR-ADMIN-API-KEY) with a key that is valid for your search service. You only have to set this header once for the duration of the session, but you will add it to every request.

PowerShell

```
$headers = @{
'api-key' = '<YOUR-ADMIN-API-KEY>'
'Content-Type' = 'application/json'
'Accept' = 'application/json' }
```

2. Create a **$url** object that specifies the service's indexes collection. Replace the service name (YOUR-SEARCH-SERVICE-NAME) with a valid search service.

PowerShell

```
$url = "https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes?
api-version=2020-06-30&`$select=name"
```

3. Run **Invoke-RestMethod** to send a GET request to the service and verify the connection. Add **ConvertTo-Json** so that you can view the responses sent back from the service.

```
PowerShell

Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json
```

If the service is empty and has no indexes, results are similar to the following example. Otherwise, you'll see a JSON representation of index definitions.

```
{
    "@odata.context":
"https://mydemo.search.windows.net/$metadata#indexes",
    "value":  [

            ]
}
```

1 - Create an index

Unless you are using the portal, an index must exist on the service before you can load data. This step defines the index and pushes it to the service. The Create Index REST API is used for this step.

Required elements of an index include a name and a fields collection. The fields collection defines the structure of a *document*. Each field has a name, type, and attributes that determine how it's used (for example, whether it is full-text searchable, filterable, or retrievable in search results). Within an index, one of the fields of type `Edm.String` must be designated as the *key* for document identity.

This index is named "hotels-quickstart" and has the field definitions you see below. It's a subset of a larger Hotels index used in other walk through articles. The field definitions have been trimmed in this quickstart for brevity.

1. Paste this example into PowerShell to create a **$body** object containing the index schema.

```
PowerShell

$body = @"
{
```

```
    "name": "hotels-quickstart",
    "fields": [
        {"name": "HotelId", "type": "Edm.String", "key": true,
"filterable": true},
        {"name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": true, "facetable": false},
        {"name": "Description", "type": "Edm.String", "searchable":
true, "filterable": false, "sortable": false, "facetable": false,
"analyzer": "en.lucene"},
        {"name": "Category", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "Tags", "type": "Collection(Edm.String)",
"searchable": true, "filterable": true, "sortable": false, "facetable":
true},
        {"name": "ParkingIncluded", "type": "Edm.Boolean",
"filterable": true, "sortable": true, "facetable": true},
        {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset",
"filterable": true, "sortable": true, "facetable": true},
        {"name": "Rating", "type": "Edm.Double", "filterable": true,
"sortable": true, "facetable": true},
        {"name": "Address", "type": "Edm.ComplexType",
        "fields": [
        {"name": "StreetAddress", "type": "Edm.String", "filterable":
false, "sortable": false, "facetable": false, "searchable": true},
        {"name": "City", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
        {"name": "StateProvince", "type": "Edm.String", "searchable":
true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "PostalCode", "type": "Edm.String", "searchable":
true, "filterable": true, "sortable": true, "facetable": true},
        {"name": "Country", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true}
        ]
     }
  ]
}
"@
```

2. Set the URI to the indexes collection on your service and the *hotels-quickstart* index.

PowerShell

```
$url = "https://<YOUR-SEARCH-
SERVICE>.search.windows.net/indexes/hotels-quickstart?api-version=2020-
06-30"
```

3. Run the command with **$url**, **$headers**, and **$body** to create the index on the service.

PowerShell

```
Invoke-RestMethod -Uri $url -Headers $headers -Method Put -Body $body |
ConvertTo-Json
```

Results should look similar to this (truncated to the first two fields for brevity):

```
{
    "@odata.context":
"https://mydemo.search.windows.net/$metadata#indexes/$entity",
    "@odata.etag":  "\"0x8D6EDE28CFEABDA\"",
    "name":  "hotels-quickstart",
    "defaultScoringProfile":  null,
    "fields":  [
                {
                    "name":  "HotelId",
                    "type":  "Edm.String",
                    "searchable":  true,
                    "filterable":  true,
                    "retrievable":  true,
                    "sortable":  true,
                    "facetable":  true,
                    "key":  true,
                    "indexAnalyzer":  null,
                    "searchAnalyzer":  null,
                    "analyzer":  null,
                    "synonymMaps":  ""
                },
                {
                    "name":  "HotelName",
                    "type":  "Edm.String",
                    "searchable":  true,
                    "filterable":  false,
                    "retrievable":  true,
                    "sortable":  true,
                    "facetable":  false,
                    "key":  false,
                    "indexAnalyzer":  null,
                    "searchAnalyzer":  null,
                    "analyzer":  null,
                    "synonymMaps":  ""
                },
. . .
```

Tip

For verification, you could also check the Indexes list in the portal.

2 - Load documents

To push documents, use a HTTP POST request to your index's URL endpoint. The REST API for this task is Add, Update, or Delete Documents.

1. Paste this example into PowerShell to create a **$body** object containing the documents you want to upload.

   This request includes two full and one partial record. The partial record demonstrates that you can upload incomplete documents. The `@search.action` parameter specifies how indexing is done. Valid values include upload, merge, mergeOrUpload, and delete. The mergeOrUpload behavior either creates a new document for hotelId = 3, or updates the contents if it already exists.

   PowerShell

```
$body = @"
{
    "value": [
    {
    "@search.action": "upload",
    "HotelId": "1",
    "HotelName": "Secret Point Motel",
    "Description": "The hotel is ideally located on the main commercial
artery of the city in the heart of New York. A few minutes away is
Time's Square and the historic centre of the city, as well as other
places of interest that make New York one of America's most attractive
and cosmopolitan cities.",
    "Category": "Boutique",
    "Tags": [ "pool", "air conditioning", "concierge" ],
    "ParkingIncluded": false,
    "LastRenovationDate": "1970-01-18T00:00:00Z",
    "Rating": 3.60,
    "Address":
        {
        "StreetAddress": "677 5th Ave",
        "City": "New York",
        "StateProvince": "NY",
        "PostalCode": "10022",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "2",
    "HotelName": "Twin Dome Motel",
    "Description": "The hotel is situated in a  nineteenth century
plaza, which has been expanded and renovated to the highest
architectural standards to create a modern, functional and first-class
hotel in which art and unique historical elements coexist with the most
modern comforts.",
```

```
    "Category": "Boutique",
    "Tags": [ "pool", "free wifi", "concierge" ],
    "ParkingIncluded": false,
    "LastRenovationDate": "1979-02-18T00:00:00Z",
    "Rating": 3.60,
    "Address":
        {
        "StreetAddress": "140 University Town Center Dr",
        "City": "Sarasota",
        "StateProvince": "FL",
        "PostalCode": "34243",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "3",
    "HotelName": "Triple Landscape Hotel",
    "Description": "The Hotel stands out for its gastronomic excellence
under the management of William Dough, who advises on and oversees all
of the Hotel’s restaurant services.",
    "Category": "Resort and Spa",
    "Tags": [ "air conditioning", "bar", "continental breakfast" ],
    "ParkingIncluded": true,
    "LastRenovationDate": "2015-09-20T00:00:00Z",
    "Rating": 4.80,
    "Address":
        {
        "StreetAddress": "3393 Peachtree Rd",
        "City": "Atlanta",
        "StateProvince": "GA",
        "PostalCode": "30326",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "4",
    "HotelName": "Sublime Cliff Hotel",
    "Description": "Sublime Cliff Hotel is located in the heart of the
historic center of Sublime in an extremely vibrant and lively area
within short walking distance to the sites and landmarks of the city
and is surrounded by the extraordinary beauty of churches, buildings,
shops and monuments. Sublime Cliff is part of a lovingly restored 1800
palace.",
    "Category": "Boutique",
    "Tags": [ "concierge", "view", "24-hour front desk service" ],
    "ParkingIncluded": true,
    "LastRenovationDate": "1960-02-06T00:00:00Z",
    "Rating": 4.60,
    "Address":
        {
        "StreetAddress": "7400 San Pedro Ave",
        "City": "San Antonio",
        "StateProvince": "TX",
```

```
        "PostalCode": "78216",
        "Country": "USA"
        }
    }
]
}
"@
```

2. Set the endpoint to the *hotels-quickstart* docs collection and include the index operation (indexes/hotels-quickstart/docs/index).

PowerShell

```
$url = "https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-quickstart/docs/index?api-version=2020-06-30"
```

3. Run the command with **$url**, **$headers**, and **$body** to load documents into the hotels-quickstart index.

PowerShell

```
Invoke-RestMethod -Uri $url -Headers $headers -Method Post -Body $body | ConvertTo-Json
```

Results should look similar to the following example. You should see a status code of 201.

```
{
    "@odata.context": "https://mydemo.search.windows.net/indexes(\u0027hotels-quickstart\u0027)/$metadata#Collection(Microsoft.Azure.Search.V2019_05_06.IndexResult)",
    "value":  [
                {
                    "key":  "1",
                    "status":  true,
                    "errorMessage":  null,
                    "statusCode":  201
                },
                {
                    "key":  "2",
                    "status":  true,
                    "errorMessage":  null,
                    "statusCode":  201
                },
                {
```

```
                    "key":  "3",
                    "status":  true,
                    "errorMessage":  null,
                    "statusCode":  201
                },
                {
                    "key":  "4",
                    "status":  true,
                    "errorMessage":  null,
                    "statusCode":  201
                }
            ]
}
```

3 - Search an index

This step shows you how to query an index using the Search Documents API.

Be sure to use single quotes on search $urls. Query strings include **$** characters, and you can omit having to escape them if the entire string is enclosed in single quotes.

1. Set the endpoint to the *hotels-quickstart* docs collection and add a **search** parameter to pass in a query string.

   This string executes an empty search (search=*), returning an unranked list (search score = 1.0) of arbitrary documents. By default, Azure Cognitive Search returns 50 matches at a time. As structured, this query returns an entire document structure and values. Add **$count=true** to get a count of all documents in the results.

   PowerShell

   ```
   $url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-quickstart/docs?api-version=2020-06-30&search=*&$count=true'
   ```

2. Run the command to send the **$url** to the service.

   PowerShell

   ```
   Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json
   ```

   Results should look similar to the following output.

```
{
"@odata.context":
"https://mydemo.search.windows.net/indexes(\u0027hotels-
quickstart\u0027)/$metadata#docs(*)",
"@odata.count":  4,
"value":  [
              {
                  "@search.score":  0.1547872,
                  "HotelId":  "2",
                  "HotelName":  "Twin Dome Motel",
                  "Description":  "The hotel is situated in a
nineteenth century plaza, which has been expanded and renovated to the
highest architectural standards to create a modern, functional and
first-class hotel in which art and unique historical elements coexist
with the most modern comforts.",
                  "Category":  "Boutique",
                  "Tags":  "pool free wifi concierge",
                  "ParkingIncluded":  false,
                  "LastRenovationDate":  "1979-02-18T00:00:00Z",
                  "Rating":  3.6,
                  "Address":  "@{StreetAddress=140 University Town
Center Dr; City=Sarasota; StateProvince=FL; PostalCode=34243;
Country=USA}"
              },
              {
                  "@search.score":  0.009068266,
                  "HotelId":  "3",
                  "HotelName":  "Triple Landscape Hotel",
                  "Description":  "The Hotel stands out for its
gastronomic excellence under the management of William Dough, who
advises on and oversees all of the Hotel\u0027s restaurant services.",
                  "Category":  "Resort and Spa",
                  "Tags":  "air conditioning bar continental
breakfast",
                  "ParkingIncluded":  true,
                  "LastRenovationDate":  "2015-09-20T00:00:00Z",
                  "Rating":  4.8,
                  "Address":  "@{StreetAddress=3393 Peachtree Rd;
City=Atlanta; StateProvince=GA; PostalCode=30326; Country=USA}"
              },
            . . .
```


Try a few other query examples to get a feel for the syntax. You can do a string search, verbatim $filter queries, limit the results set, scope the search to specific fields, and more.

PowerShell

```
# Query example 1
# Search the entire index for the terms 'restaurant' and 'wifi'
# Return only the HotelName, Description, and Tags fields
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-
```

```
quickstart/docs?api-version=2020-06-30&search=restaurant
wifi&$count=true&$select=HotelName,Description,Tags'

# Query example 2
# Apply a filter to the index to find hotels rated 4 or higher
# Returns the HotelName and Rating. Two documents match.
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-
quickstart/docs?api-version=2020-06-30&search=*&$filter=Rating gt
4&$select=HotelName,Rating'

# Query example 3
# Take the top two results, and show only HotelName and Category in the
results
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-
quickstart/docs?api-version=2020-06-
30&search=boutique&$top=2&$select=HotelName,Category'

# Query example 4
# Sort by a specific field (Address/City) in ascending order

$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/hotels-
quickstart/docs?api-version=2020-06-30&search=pool&$orderby=Address/City
asc&$select=HotelName, Address/City, Tags, Rating'
```

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you are using a free service, remember that you are limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

In this quickstart, you used PowerShell to step through the basic workflow for creating and accessing content in Azure Cognitive Search. With the concepts in mind, we recommend moving on to more advanced scenarios, such as indexing from Azure data sources;

REST Tutorial: Index and search semi-structured data (JSON blobs) in Azure Cognitive Search

Additional resources

Documentation

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Autocomplete (Azure Cognitive Search REST API)

Send queries and obtain results based on partial inputs.

List Indexes (Azure Cognitive Search REST API)

Return a list of index names for indexes in the current Azure Cognitive Search service.

Supported data types (Azure Cognitive Search REST API)

Allowed data types for fields in an Azure Cognitive Search index.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Index operations using Azure Cognitive Search REST APIs

Learn which REST APIs are used to create, delete, and update Azure Cognitive Search index definitions.

Show 5 more

Quickstart: Create an Azure Cognitive Search index in Python using Jupyter Notebook

Article • 01/29/2023 • 9 minutes to read

In this exercise, build a Jupyter Notebook that creates, loads, and queries an Azure Cognitive Search index using Python and the azure-search-documents library in the Azure SDK for Python. This article explains how to build a notebook step by step. Alternatively, you can download and run a finished Jupyter Python notebook.

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

The following services and tools are used in this quickstart.

- Visual Studio Code with the Python extension (or equivalent tool), with Python 3.7 or later
- azure-search-documents package from the Azure SDK for Python
- Create a search service or find an existing service under your current subscription. You can use the Free tier for this quickstart.

Copy a key and URL

To connect to your search service, provide the endpoint and an access key. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

You'll provide these values in the next section when you set up the connection.

Connect to Azure Cognitive Search

In this task, create the notebook, load the libraries, and set up your clients.

1. Create a new Python3 notebook in Visual Studio Code:
   a. Press F1 and search for "Python Select Interpreter" and choose a version of Python 3.7 or later.
   b. Press F1 again and search for "Create: New Jupyter Notebook". You should have an empty, untitled `.ipynb` file open in the editor ready for the first entry.

2. In the first cell, load the libraries from the Azure SDK for Python, including azure-search-documents.

Python

```
%pip install azure-search-documents --pre
%pip show azure-search-documents

import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents import SearchClient
from azure.search.documents.indexes.models import (
    ComplexField,
    CorsOptions,
    SearchIndex,
    ScoringProfile,
    SearchFieldDataType,
```

```
    SimpleField,
    SearchableField
)
```

3. Add a second cell and paste in the connection information. This cell also sets up the clients you'll use for specific operations: SearchIndexClient to create an index, and SearchClient to query an index.

Python

```python
service_name = "YOUR-SEARCH-SERVICE-NAME"
admin_key = "YOUR-SEARCH-SERVICE-ADMIN-API-KEY"

index_name = "hotels-quickstart"

# Create an SDK client
endpoint = "https://{}.search.windows.net/".format(service_name)
admin_client = SearchIndexClient(endpoint=endpoint,
                      index_name=index_name,
                      credential=AzureKeyCredential(admin_key))

search_client = SearchClient(endpoint=endpoint,
                      index_name=index_name,
                      credential=AzureKeyCredential(admin_key))
```

4. In the third cell, run a delete_index operation to clear your service of any existing *hotels-quickstart* indexes. Deleting the index allows you to create another *hotels-quickstart* index of the same name.

Python

```python
try:
    result = admin_client.delete_index(index_name)
    print ('Index', index_name, 'Deleted')
except Exception as ex:
    print (ex)
```

5. Run each step.

1 - Create an index

Required index elements include a name, a fields collection, and a document key that uniquely identifies each search document. The fields collection defines the structure of a logical *search document*, used for both loading data and returning results.

Within the fields collection, each field has a name, type, and attributes that determine how the field is used (for example, whether it's full-text searchable, filterable, or retrievable in search results). Within an index, one of the fields of type `Edm.String` must be designated as the *key* for document identity.

This index is named "hotels-quickstart" and has the field definitions you see below. It's a subset of a larger Hotels index used in other walkthroughs. We trimmed it in this quickstart for brevity.

1. In the next cell, paste the following example into a cell to provide the schema.

Python

```
# Specify the index schema
name = index_name
fields = [
        SimpleField(name="HotelId", type=SearchFieldDataType.String,
key=True),
        SearchableField(name="HotelName",
type=SearchFieldDataType.String, sortable=True),
        SearchableField(name="Description",
type=SearchFieldDataType.String, analyzer_name="en.lucene"),
        SearchableField(name="Description_fr",
type=SearchFieldDataType.String, analyzer_name="fr.lucene"),
        SearchableField(name="Category",
type=SearchFieldDataType.String, facetable=True, filterable=True,
sortable=True),

        SearchableField(name="Tags", collection=True,
type=SearchFieldDataType.String, facetable=True, filterable=True),

        SimpleField(name="ParkingIncluded",
type=SearchFieldDataType.Boolean, facetable=True, filterable=True,
sortable=True),
        SimpleField(name="LastRenovationDate",
type=SearchFieldDataType.DateTimeOffset, facetable=True,
filterable=True, sortable=True),
        SimpleField(name="Rating", type=SearchFieldDataType.Double,
facetable=True, filterable=True, sortable=True),

        ComplexField(name="Address", fields=[
            SearchableField(name="StreetAddress",
type=SearchFieldDataType.String),
            SearchableField(name="City",
type=SearchFieldDataType.String, facetable=True, filterable=True,
sortable=True),
            SearchableField(name="StateProvince",
type=SearchFieldDataType.String, facetable=True, filterable=True,
sortable=True),
            SearchableField(name="PostalCode",
type=SearchFieldDataType.String, facetable=True, filterable=True,
sortable=True),
```

```
            SearchableField(name="Country",
type=SearchFieldDataType.String, facetable=True, filterable=True,
sortable=True),
        ])
    ]
cors_options = CorsOptions(allowed_origins=["*"],
max_age_in_seconds=60)
scoring_profiles = []
suggester = [{'name': 'sg', 'source_fields': ['Tags', 'Address/City',
'Address/Country']}]
```

2. In another cell, formulate the request. This create_index request targets the indexes collection of your search service and creates a SearchIndex based on the index schema you provided in the previous cell.

Python

```
index = SearchIndex(
    name=name,
    fields=fields,
    scoring_profiles=scoring_profiles,
    suggesters = suggester,
    cors_options=cors_options)

try:
    result = admin_client.create_index(index)
    print ('Index', result.name, 'created')
except Exception as ex:
    print (ex)
```

3. Run each step.

2 - Load documents

To load documents, create a documents collection, using an index action for the operation type (upload, merge-and-upload, and so forth). Documents originate from HotelsData on GitHub.

1. In a new cell, provide four documents that conform to the index schema. Specify an upload action for each document.

Python

```
documents = [
    {
    "@search.action": "upload",
    "HotelId": "1",
    "HotelName": "Secret Point Motel",
```

```
    "Description": "The hotel is ideally located on the main commercial
artery of the city in the heart of New York. A few minutes away is
Time's Square and the historic centre of the city, as well as other
places of interest that make New York one of America's most attractive
and cosmopolitan cities.",
    "Description_fr": "L'hôtel est idéalement situé sur la principale
artère commerciale de la ville en plein cœur de New York. A quelques
minutes se trouve la place du temps et le centre historique de la
ville, ainsi que d'autres lieux d'intérêt qui font de New York l'une
des villes les plus attractives et cosmopolites de l'Amérique.",
    "Category": "Boutique",
    "Tags": [ "pool", "air conditioning", "concierge" ],
    "ParkingIncluded": "false",
    "LastRenovationDate": "1970-01-18T00:00:00Z",
    "Rating": 3.60,
    "Address": {
        "StreetAddress": "677 5th Ave",
        "City": "New York",
        "StateProvince": "NY",
        "PostalCode": "10022",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "2",
    "HotelName": "Twin Dome Motel",
    "Description": "The hotel is situated in a  nineteenth century
plaza, which has been expanded and renovated to the highest
architectural standards to create a modern, functional and first-class
hotel in which art and unique historical elements coexist with the most
modern comforts.",
    "Description_fr": "L'hôtel est situé dans une place du XIXe siècle,
qui a été agrandie et rénovée aux plus hautes normes architecturales
pour créer un hôtel moderne, fonctionnel et de première classe dans
lequel l'art et les éléments historiques uniques coexistent avec le
confort le plus moderne.",
    "Category": "Boutique",
    "Tags": [ "pool", "free wifi", "concierge" ],
    "ParkingIncluded": "false",
    "LastRenovationDate": "1979-02-18T00:00:00Z",
    "Rating": 3.60,
    "Address": {
        "StreetAddress": "140 University Town Center Dr",
        "City": "Sarasota",
        "StateProvince": "FL",
        "PostalCode": "34243",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "3",
    "HotelName": "Triple Landscape Hotel",
    "Description": "The Hotel stands out for its gastronomic excellence
```

```
under the management of William Dough, who advises on and oversees all
of the Hotel's restaurant services.",
    "Description_fr": "L'hôtel est situé dans une place du XIXe siècle,
qui a été agrandie et rénovée aux plus hautes normes architecturales
pour créer un hôtel moderne, fonctionnel et de première classe dans
lequel l'art et les éléments historiques uniques coexistent avec le
confort le plus moderne.",
    "Category": "Resort and Spa",
    "Tags": [ "air conditioning", "bar", "continental breakfast" ],
    "ParkingIncluded": "true",
    "LastRenovationDate": "2015-09-20T00:00:00Z",
    "Rating": 4.80,
    "Address": {
        "StreetAddress": "3393 Peachtree Rd",
        "City": "Atlanta",
        "StateProvince": "GA",
        "PostalCode": "30326",
        "Country": "USA"
        }
    },
    {
    "@search.action": "upload",
    "HotelId": "4",
    "HotelName": "Sublime Cliff Hotel",
    "Description": "Sublime Cliff Hotel is located in the heart of the
historic center of Sublime in an extremely vibrant and lively area
within short walking distance to the sites and landmarks of the city
and is surrounded by the extraordinary beauty of churches, buildings,
shops and monuments. Sublime Cliff is part of a lovingly restored 1800
palace.",
    "Description_fr": "Le sublime Cliff Hotel est situé au coeur du
centre historique de sublime dans un quartier extrêmement animé et
vivant, à courte distance de marche des sites et monuments de la ville
et est entouré par l'extraordinaire beauté des églises, des bâtiments,
des commerces et Monuments. Sublime Cliff fait partie d'un Palace 1800
restauré avec amour.",
    "Category": "Boutique",
    "Tags": [ "concierge", "view", "24-hour front desk service" ],
    "ParkingIncluded": "true",
    "LastRenovationDate": "1960-02-06T00:00:00Z",
    "Rating": 4.60,
    "Address": {
        "StreetAddress": "7400 San Pedro Ave",
        "City": "San Antonio",
        "StateProvince": "TX",
        "PostalCode": "78216",
        "Country": "USA"
        }
    }
]
```


2. In another cell, formulate the request. This upload_documents request targets the docs collection of the hotels-quickstart index and pushes the documents provided

in the previous step into the Cognitive Search index.

```python
try:
    result = search_client.upload_documents(documents=documents)
    print("Upload of new document succeeded:
{}".format(result[0].succeeded))
except Exception as ex:
    print (ex.message)
```

3. Run each step to push the documents to an index in your search service.

3 - Search an index

This step shows you how to query an index using the **search** method of the search.client class.

1. The following step executes an empty search (`search=*`), returning an unranked list (search score = 1.0) of arbitrary documents. Because there are no criteria, all documents are included in results. This query prints just two of the fields in each document. It also adds `include_total_count=True` to get a count of all documents (4) in the results.

```python
results = search_client.search(search_text="*",
include_total_count=True)

print ('Total Documents Matching Query:', results.get_count())
for result in results:
    print("{}: {}".format(result["HotelId"], result["HotelName"]))
```

2. The next query adds whole terms to the search expression ("wifi"). This query specifies that results contain only those fields in the `select` statement. Limiting the fields that come back minimizes the amount of data sent back over the wire and reduces search latency.

```python
results = search_client.search(search_text="wifi",
include_total_count=True, select='HotelId,HotelName,Tags')

print ('Total Documents Matching Query:', results.get_count())
for result in results:
```

```
    print("{}: {}: {}".format(result["HotelId"], result["HotelName"],
result["Tags"]))
```

3. Next, apply a filter expression, returning only those hotels with a rating greater than 4, sorted in descending order.

Python

```
results = search_client.search(search_text="hotels",
select='HotelId,HotelName,Rating', filter='Rating gt 4',
order_by='Rating desc')

for result in results:
    print("{}: {} - {} rating".format(result["HotelId"],
result["HotelName"], result["Rating"]))
```

4. Add `search_fields` (an array) to scope query matching to a single field.

Python

```
results = search_client.search(search_text="sublime", search_fields=
['HotelName'], select='HotelId,HotelName')

for result in results:
    print("{}: {}".format(result["HotelId"], result["HotelName"]))
```

5. Facets are labels that can be used to compose facet navigation structure. This query returns facets and counts for Category.

Python

```
results = search_client.search(search_text="*", facets=["Category"])

facets = results.get_facets()

for facet in facets["Category"]:
    print("    {}".format(facet))
```

6. In this example, look up a specific document based on its key. You would typically want to return a document when a user selects a document in a search result.

Python

```
result = search_client.get_document(key="3")

print("Details for hotel '3' are:")
print("Name: {}".format(result["HotelName"]))
```

```
print("Rating: {}".format(result["Rating"]))
print("Category: {}".format(result["Category"]))
```

7. In the last example, we'll use the autocomplete function. Autocomplete is typically used in a search box to provide potential matches as the user types into the search box.

   When the index was created, a suggester named `sg` was also created as part of the request. A suggester definition specifies which fields can be used to find potential matches to suggester requests. In this example, those fields are 'Tags', 'Address/City', 'Address/Country'. To simulate auto-complete, pass in the letters "sa" as a partial string. The autocomplete method of SearchClient sends back potential term matches.

   Python

   ```
   search_suggestion = 'sa'
   results = search_client.autocomplete(search_text=search_suggestion,
   suggester_name="sg", mode='twoTerms')

   print("Autocomplete for:", search_suggestion)
   for result in results:
       print (result['text'])
   ```

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

In this Python quickstart, you worked through the fundamental workflow using the azure.search.documents library from the Python SDK. You performed tasks that created an index, loaded it with documents, and ran queries. To continue learning, try the following tutorial.

Tutorial: Add search to web apps

Additional resources

Documentation

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Azure Cognitive Search client library for Python

Python tutorial: AI on Azure blobs - Azure Cognitive Search

Step through an example of text extraction and natural language processing over content in Blob storage using a Jupyter Python notebook and the Azure Cognitive Search REST APIs.

azure-search-documents package

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Custom AML skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets with Azure Machine Learning models.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Show 5 more

Quickstart: Deploy Cognitive Search using Bicep

Article • 04/13/2023

This article walks you through the process for using a Bicep file to deploy an Azure Cognitive Search resource in the Azure portal.

Bicep is a domain-specific language (DSL) that uses declarative syntax to deploy Azure resources. It provides concise syntax, reliable type safety, and support for code reuse. Bicep offers the best authoring experience for your infrastructure-as-code solutions in Azure.

Only those properties included in the template are used in the deployment. If more customization is required, such as setting up network security, you can update the service as a post-deployment task. To customize an existing service with the fewest steps, use Azure CLI or Azure PowerShell. If you're evaluating preview features, use the Management REST API.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

Review the Bicep file

The Bicep file used in this quickstart is from Azure Quickstart Templates.

Bicep

```
@description('Service name must only contain lowercase letters, digits or dashes, cannot use dash as the first two or last one characters, cannot contain consecutive dashes, and is limited between 2 and 60 characters in length.')
@minLength(2)
@maxLength(60)
param name string

@allowed([
  'free'
  'basic'
  'standard'
  'standard2'
  'standard3'
  'storage_optimized_l1'
```

```
  'storage_optimized_l2'
])
@description('The pricing tier of the search service you want to create (for
example, basic or standard).')
param sku string = 'standard'

@description('Replicas distribute search workloads across the service. You
need at least two replicas to support high availability of query workloads
(not applicable to the free tier).')
@minValue(1)
@maxValue(12)
param replicaCount int = 1

@description('Partitions allow for scaling of document count as well as
faster indexing by sharding your index over multiple search units.')
@allowed([
  1
  2
  3
  4
  6
  12
])
param partitionCount int = 1

@description('Applicable only for SKUs set to standard3. You can set this
property to enable a single, high density partition that allows up to 1000
indexes, which is much higher than the maximum indexes allowed for any other
SKU.')
@allowed([
  'default'
  'highDensity'
])
param hostingMode string = 'default'

@description('Location for all resources.')
param location string = resourceGroup().location

resource search 'Microsoft.Search/searchServices@2020-08-01' = {
  name: name
  location: location
  sku: {
    name: sku
  }
  properties: {
    replicaCount: replicaCount
    partitionCount: partitionCount
    hostingMode: hostingMode
  }
}
```

The Azure resource defined in this Bicep file:

- Microsoft.Search/searchServices: create an Azure Cognitive Search service

Deploy the Bicep file

1. Save the Bicep file as **main.bicep** to your local computer.

2. Deploy the Bicep file using either Azure CLI or Azure PowerShell.

CLI

Azure CLI

```
az group create --name exampleRG --location eastus
az deployment group create --resource-group exampleRG --template-
file main.bicep --parameters serviceName=<service-name>
```

> **Note**
>
> Replace **<service-name>** with the name of the Search service. The service name must only contain lowercase letters, digits, or dashes. You can't use a dash as the first two characters or the last character. The name has a minimum length of 2 characters and a maximum length of 60 characters.

When the deployment finishes, you should see a message indicating the deployment succeeded.

Review deployed resources

Use the Azure portal, Azure CLI, or Azure PowerShell to list the deployed resources in the resource group.

CLI

Azure CLI

```
az resource list --resource-group exampleRG
```

Clean up resources

Other Cognitive Search quickstarts and tutorials build upon this quickstart. If you plan to continue on to work with subsequent quickstarts and tutorials, you may wish to leave this resource in place. When no longer needed, use the Azure portal, Azure CLI, or Azure PowerShell to delete the resource group and its resources.

CLI

Azure CLI

```
az group delete --name exampleRG
```

Next steps

In this quickstart, you created a Cognitive Search service using a Bicep file, and then validated the deployment. To learn more about Cognitive Search and Azure Resource Manager, continue on to the articles below.

- Read an overview of Azure Cognitive Search.
- Create an index for your search service.
- Create a demo app using the portal wizard.
- Create a skillset to extract information from your data.

Quickstart: Deploy Cognitive Search using an Azure Resource Manager template

Article • 04/13/2023

This article walks you through the process for using an Azure Resource Manager (ARM) template to deploy an Azure Cognitive Search resource in the Azure portal.

A resource manager template is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax. In declarative syntax, you describe your intended deployment without writing the sequence of programming commands to create the deployment.

Only those properties included in the template are used in the deployment. If more customization is required, such as setting up network security, you can update the service as a post-deployment task. To customize an existing service with the fewest steps, use Azure CLI or Azure PowerShell. If you're evaluating preview features, use the Management REST API.

Assuming your environment meets the prerequisites and you're familiar with using ARM templates, select the **Deploy to Azure** button. The template will open in the Azure portal.

Deploy to Azure

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

Review the template

The template used in this quickstart is from Azure Quickstart Templates.

JSON

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "metadata": {
```

```
    "_generator": {
      "name": "bicep",
      "version": "0.5.6.12127",
      "templateHash": "11257266040777038564"
    }
  },
  "parameters": {
    "name": {
      "type": "string",
      "maxLength": 60,
      "minLength": 2,
      "metadata": {
        "description": "Service name must only contain lowercase letters,
digits or dashes, cannot use dash as the first two or last one characters,
cannot contain consecutive dashes, and is limited between 2 and 60
characters in length."
      }
    },
    "sku": {
      "type": "string",
      "defaultValue": "standard",
      "metadata": {
        "description": "The pricing tier of the search service you want to
create (for example, basic or standard)."
      },
      "allowedValues": [
        "free",
        "basic",
        "standard",
        "standard2",
        "standard3",
        "storage_optimized_l1",
        "storage_optimized_l2"
      ]
    },
    "replicaCount": {
      "type": "int",
      "defaultValue": 1,
      "maxValue": 12,
      "minValue": 1,
      "metadata": {
        "description": "Replicas distribute search workloads across the
service. You need at least two replicas to support high availability of
query workloads (not applicable to the free tier)."
      }
    },
    "partitionCount": {
      "type": "int",
      "defaultValue": 1,
      "allowedValues": [
        1,
        2,
        3,
        4,
        6,
```

```
        12
      ],
      "metadata": {
        "description": "Partitions allow for scaling of document count as
well as faster indexing by sharding your index over multiple search units."
      }
    },
    "hostingMode": {
      "type": "string",
      "defaultValue": "default",
      "allowedValues": [
        "default",
        "highDensity"
      ],
      "metadata": {
        "description": "Applicable only for SKUs set to standard3. You can
set this property to enable a single, high density partition that allows up
to 1000 indexes, which is much higher than the maximum indexes allowed for
any other SKU."
      }
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]",
      "metadata": {
        "description": "Location for all resources."
      }
    }
  },
  "resources": [
    {
      "type": "Microsoft.Search/searchServices",
      "apiVersion": "2020-08-01",
      "name": "[parameters('name')]",
      "location": "[parameters('location')]",
      "sku": {
        "name": "[parameters('sku')]"
      },
      "properties": {
        "replicaCount": "[parameters('replicaCount')]",
        "partitionCount": "[parameters('partitionCount')]",
        "hostingMode": "[parameters('hostingMode')]"
      }
    }
  ]
}
```

The Azure resource defined in this template:

- Microsoft.Search/searchServices: create an Azure Cognitive Search service

Deploy the template

Select the following image to sign in to Azure and open a template. The template creates an Azure Cognitive Search resource.

Deploy to Azure

The portal displays a form that allows you to easily provide parameter values. Some parameters are pre-filled with the default values from the template. You will need to provide your subscription, resource group, location, and service name. If you want to use Cognitive Services in an AI enrichment pipeline, for example to analyze binary image files for text, choose a location that offers both Cognitive Search and Cognitive Services. Both services are required to be in the same region for AI enrichment workloads. Once you have completed the form, you will need to agree to the terms and conditions and then select the purchase button to complete your deployment.

Review deployed resources

When your deployment is complete you can access your new resource group and new search service in the portal.

Clean up resources

Other Cognitive Search quickstarts and tutorials build upon this quickstart. If you plan to continue on to work with subsequent quickstarts and tutorials, you may wish to leave this resource in place. When no longer needed, you can delete the resource group, which deletes the Cognitive Search service and related resources.

Next steps

In this quickstart, you created a Cognitive Search service using an ARM template, and validated the deployment. To learn more about Cognitive Search and Azure Resource Manager, continue on to the articles below.

- Read an overview of Azure Cognitive Search.
- Create an index for your search service.
- Create a demo app using the portal wizard.
- Create a skillset to extract information from your data.

Quickstart: Create an Azure Cognitive Search service using Terraform

Article • 04/14/2023

This article shows how to use Terraform to create an Azure Cognitive Search service using Terraform.

Terraform enables the definition, preview, and deployment of cloud infrastructure. Using Terraform, you create configuration files using HCL syntax. The HCL syntax allows you to specify the cloud provider - such as Azure - and the elements that make up your cloud infrastructure. After you create your configuration files, you create an *execution plan* that allows you to preview your infrastructure changes before they're deployed. Once you verify the changes, you apply the execution plan to deploy the infrastructure.

In this article, you learn how to:

- ✓ Create a random pet name for the Azure resource group name using random_pet
- ✓ Create an Azure resource group using azurerm_resource_group
- ✓ Create a random string using random_string
- ✓ Create an Azure Cognitive Search service using azurerm_search_service

> **Note**
>
> This article was partially created with the help of artificial intelligence. Before publishing, an author reviewed and revised the content as needed. See **Our principles for using AI-generated content in Microsoft Learn**.

Prerequisites

- Install and configure Terraform

Implement the Terraform code

> **Note**
>
> The sample code for this article is located in the **Azure Terraform GitHub repo**. You can view the log file containing the **test results from current and previous**

versions of Terraform.

See more articles and sample code showing how to use Terraform to manage Azure resources

1. Create a directory in which to test and run the sample Terraform code and make it the current directory.

2. Create a file named `main.tf` and insert the following code:

Terraform

```
resource "random_pet" "rg_name" {
  prefix = var.resource_group_name_prefix
}

resource "azurerm_resource_group" "rg" {
  name     = random_pet.rg_name.id
  location = var.resource_group_location
}

resource "random_string" "azurerm_search_service_name" {
  length  = 25
  upper   = false
  numeric = false
  special = false
}

resource "azurerm_search_service" "search" {
  name                =
random_string.azurerm_search_service_name.result
  resource_group_name = azurerm_resource_group.rg.name
  location            = azurerm_resource_group.rg.location
  sku                 = var.sku
  replica_count       = var.replica_count
  partition_count     = var.partition_count
}
```

3. Create a file named `outputs.tf` and insert the following code:

Terraform

```
output "resource_group_name" {
  value = azurerm_resource_group.rg.name
}

output "azurerm_search_service_name" {
  value = azurerm_search_service.search.name
}
```

4. Create a file named `providers.tf` and insert the following code:

Terraform

```
terraform {
  required_version = ">=1.0"
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~>3.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~>3.0"
    }
  }
}
provider "azurerm" {
  features {}
}
```

5. Create a file named `variables.tf` and insert the following code:

Terraform

```
variable "resource_group_location" {
  type        = string
  description = "Location for all resources."
  default     = "eastus"
}

variable "resource_group_name_prefix" {
  type        = string
  description = "Prefix of the resource group name that's combined with
a random ID so name is unique in your Azure subscription."
  default     = "rg"
}

variable "sku" {
  description = "The pricing tier of the search service you want to
create (for example, basic or standard)."
  default     = "standard"
  type        = string
  validation {
    condition     = contains(["free", "basic", "standard", "standard2",
"standard3", "storage_optimized_l1", "storage_optimized_l2"], var.sku)
    error_message = "The sku must be one of the following values: free,
basic, standard, standard2, standard3, storage_optimized_l1,
storage_optimized_l2."
  }
}
```

```
variable "replica_count" {
  type        = number
  description = "Replicas distribute search workloads across the
service. You need at least two replicas to support high availability of
query workloads (not applicable to the free tier)."
  default     = 1
  validation {
    condition     = var.replica_count >= 1 && var.replica_count <= 12
    error_message = "The replica_count must be between 1 and 12."
  }
}

variable "partition_count" {
  type        = number
  description = "Partitions allow for scaling of document count as well
as faster indexing by sharding your index over multiple search units."
  default     = 1
  validation {
    condition     = contains([1, 2, 3, 4, 6, 12], var.partition_count)
    error_message = "The partition_count must be one of the following
values: 1, 2, 3, 4, 6, 12."
  }
}
```

Initialize Terraform

Run terraform init to initialize the Terraform deployment. This command downloads the Azure provider required to manage your Azure resources.

Console

```
terraform init -upgrade
```

Key points:

- The `-upgrade` parameter upgrades the necessary provider plugins to the newest version that complies with the configuration's version constraints.

Create a Terraform execution plan

Run terraform plan to create an execution plan.

Console

```
terraform plan -out main.tfplan
```

Key points:

- The `terraform plan` command creates an execution plan, but doesn't execute it. Instead, it determines what actions are necessary to create the configuration specified in your configuration files. This pattern allows you to verify whether the execution plan matches your expectations before making any changes to actual resources.
- The optional `-out` parameter allows you to specify an output file for the plan. Using the `-out` parameter ensures that the plan you reviewed is exactly what is applied.
- To read more about persisting execution plans and security, see the security warning section.

Apply a Terraform execution plan

Run terraform apply to apply the execution plan to your cloud infrastructure.

Console

```
terraform apply main.tfplan
```

Key points:

- The `terraform apply` command above assumes you previously ran `terraform plan -out main.tfplan`.
- If you specified a different filename for the `-out` parameter, use that same filename in the call to `terraform apply`.
- If you didn't use the `-out` parameter, call `terraform apply` without any parameters.

Verify the results

1. Get the Azure resource name in which the Azure Cognitive Search service was created.

   Console

   ```
   resource_group_name=$(terraform output -raw resource_group_name)
   ```

2. Get the Azure Cognitive Search service name.

   Console

```
azurerm_search_service_name=$(terraform output -raw
azurerm_search_service_name)
```

3. Run az search service show to show the Azure Cognitive Search service you created in this article.

Azure CLI

```
az search service show --name $azurerm_search_service_name \
                       --resource-group $resource_group_name
```

Clean up resources

When you no longer need the resources created via Terraform, do the following steps:

1. Run terraform plan and specify the `destroy` flag.

Console

```
terraform plan -destroy -out main.destroy.tfplan
```

Key points:

- The `terraform plan` command creates an execution plan, but doesn't execute it. Instead, it determines what actions are necessary to create the configuration specified in your configuration files. This pattern allows you to verify whether the execution plan matches your expectations before making any changes to actual resources.
- The optional `-out` parameter allows you to specify an output file for the plan. Using the `-out` parameter ensures that the plan you reviewed is exactly what is applied.
- To read more about persisting execution plans and security, see the security warning section.

2. Run terraform apply to apply the execution plan.

Console

```
terraform apply main.destroy.tfplan
```

Troubleshoot Terraform on Azure

Troubleshoot common problems when using Terraform on Azure

Next steps

Create an Azure Cognitive Search index using the Azure portal

1 - Overview of adding search to a website with .NET

Article • 11/23/2022 • 2 minutes to read

This tutorial builds a website to search through a catalog of books then deploys the website to an Azure Static Web App.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

What does the sample do?

This sample website provides access to a catalog of 10,000 books. A user can search the catalog by entering text in the search bar. While the user enters text, the website uses the search index's suggest feature to complete the text. Once the query finishes, the list of books is displayed with a portion of the details. A user can select a book to see all the details, stored in the search index, of the book.

The search experience includes:

- Search – provides search functionality for the application.
- Suggest – provides suggestions as the user is typing in the search bar.
- Document Lookup – looks up a document by ID to retrieve all of its contents for the details page.

How is the sample organized?

The sample includes the following:

App	Purpose	GitHub Repository Location
Client	React app (presentation layer) to display books, with search. It calls the Azure Function app.	/search-website-functions-v4/client
Server	Azure .NET Function app (business layer) - calls the Azure Cognitive Search API using .NET SDK	/search-website-functions-v4/api
Bulk insert	.NET file to create the index and add documents to it.	/search-website-functions-v4/bulk-insert

Set up your development environment

Install the following for your local development environment.

- .NET 6
- Git
- Visual Studio Code and the following extensions
  - Azure Static Web App
- Optional:
  - This tutorial doesn't run the Azure Function API locally but if you intend to run it locally, you need to install azure-functions-core-tools.

Fork and clone the search sample with git

Forking the sample repository is critical to be able to deploy the Static Web App. The web apps determine the build actions and deployment content based on your own GitHub fork location. Code execution in the Static Web App is remote, with Azure Static Web Apps reading from the code in your forked sample.

1. On GitHub, fork the sample repository.

   Complete the fork process in your web browser with your GitHub account. This tutorial uses your fork as part of the deployment to an Azure Static Web App.

2. At a Bash terminal, download your forked sample application to your local computer.

Replace `YOUR-GITHUB-ALIAS` with your GitHub alias.

```bash
git clone https://github.com/YOUR-GITHUB-ALIAS/azure-search-dotnet-samples
```

3. At the same Bash terminal, go into your forked repository for this website search example:

```bash
cd azure-search-dotnet-samples
```

4. Use the Visual Studio Code command, `code .` to open your forked repository. The remaining tasks are accomplished from Visual Studio Code, unless specified.

```bash
code .
```

Create a resource group for your Azure resources

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the **Resources** section, select Add (+), and then select **Create Resource Group**.

3. Enter a resource group name, such as `cognitive-search-website-tutorial`.

4. Select a location close to you.

5. When you create the Cognitive Search and Static Web Apps resources, later in the tutorial, use this resource group.

   Creating a resource group gives you a logical unit to manage the resources, including deleting them when you're finished using them.

Next steps

- Create a Search Index and load with documents
- Deploy your Static Web App

Additional resources

🕮 Documentation

Add language analyzers to string fields - Azure Cognitive Search

Multi-lingual lexical analysis for non-English queries and indexes in Azure Cognitive Search.

.NET tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with .NET.

Full text query and indexing engine architecture (Lucene) - Azure Cognitive Search

Explore Lucene query processing and document retrieval concepts for full text search, as related to Azure Cognitive Search.

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure Cognitive Search index.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

.NET tutorial: Search integration highlights - Azure Cognitive Search

Understand the .NET SDK Search integration queries used in the Search-enabled website with this cheat sheet.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Create a suggester - Azure Cognitive Search

Enable type-ahead query actions in Azure Cognitive Search by creating suggesters and formulating requests that invoke autocomplete or autosuggested query terms.

Show 5 more

Training

Learning path

Azure Static Web Apps - Training

Azure Static Web Apps is a modern web app service that offers streamlined full-stack development from source code to global high availability.

2 - Create and load Search Index with .NET

Article • 01/23/2023 • 4 minutes to read

Continue to build your search-enabled website by following these steps:

- Create a search resource
- Create a new index
- Import data with .NET using the sample script and Azure SDK Azure.Search.Documents.

Create an Azure Cognitive Search resource

Create a new search resource using PowerShell and the **Az.Search** module. In this section, you'll also create a query key used for read-access to the index, and get the built-in admin key used for adding objects.

1. In Visual Studio Code, open a new terminal window.

2. Connect to Azure:

   PowerShell

   ```
   Connect-AzAccount -TenantID <your-tenant-ID>
   ```

   > **Note**
   >
   > You might need to provide a tenant ID, which you can find in the Azure portal in **Portal settings** > **Directories + subscriptions**.

3. Before creating a new search service, you can list existing search services for your subscription to see if there's one you want to use:

   PowerShell

   ```
   Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
   ```

4. Load the **Az.Search** module:

   PowerShell

```
Install-Module -Name Az.Search
```

5. Create a new search service. Use the following cmdlet as a template, substituting valid values for the resource group, service name, tier, region, partitions, and replicas:

PowerShell

```
New-AzSearchService -ResourceGroupName "my resource group"  -Name
"myDemoSearchSvc" -Sku "Free" -Location "West US" -PartitionCount 1 -
ReplicaCount 1 -HostingMode Default
```

Prompt	Enter
Enter a globally unique name for the new search service.	**Remember this name**. This resource name becomes part of your resource endpoint.
Select a resource group for new resources	Use the resource group you created for this tutorial.
Select the SKU for your search service.	Use **Free** for this tutorial. You can't change a SKU pricing tier after the service is created.
Select a location for new resources.	Select a region close to you.

6. Create a query key that grants read access to a search service. Query keys have to be explicitly created. Copy the query key to Notepad so that you can paste it into the client code in a later step:

PowerShell

```
New-AzSearchQueryKey -ResourceGroupName "my resource group"  -
ServiceName "myDemoSearchSvc" -Name "mySrchQueryKey"
```

7. Get the search service admin API key that was automatically created for your search service. An admin API key provides write access to the search service. Copy either one of the admin keys to Notepad so that you can use it in the bulk import step that creates and loads an index:

PowerShell

```
Get-AzSearchAdminKeyPair  -ResourceGroupName "my resource group" -
ServiceName "myDemoSearchSvc"
```

Prepare the bulk import script for Search

The script uses the Azure SDK for Cognitive Search:

- NuGet package Azure.Search.Documents
- Reference Documentation

1. In Visual Studio Code, open the `Program.cs` file in the subdirectory, `search-website-functions-v4/bulk-insert`, replace the following variables with your own values to authenticate with the Azure Search SDK:

   - YOUR-SEARCH-RESOURCE-NAME
   - YOUR-SEARCH-ADMIN-KEY

   C#

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using AzureSearch.BulkInsert;
using ServiceStack;

const string BOOKS_URL =
"https://raw.githubusercontent.com/zygmuntz/goodbooks-
10k/master/books.csv";
const string SEARCH_ENDPOINT = "https://YOUR-SEARCH-RESOURCE-
NAME.search.windows.net";
const string SEARCH_KEY = "YOUR-SEARCH-ADMIN-KEY";
const string SEARCH_INDEX_NAME = "good-books";

Uri searchEndpointUri = new(SEARCH_ENDPOINT);

SearchClient client = new(
    searchEndpointUri,
    SEARCH_INDEX_NAME,
    new AzureKeyCredential(SEARCH_KEY));

SearchIndexClient clientIndex = new(
    searchEndpointUri,
    new AzureKeyCredential(SEARCH_KEY));

await CreateIndexAsync(clientIndex);
await BulkInsertAsync(client);

static async Task CreateIndexAsync(SearchIndexClient clientIndex)
{
    Console.WriteLine("Creating (or updating) search index");
    SearchIndex index = new BookSearchIndex(SEARCH_INDEX_NAME);
    var result = await clientIndex.CreateOrUpdateIndexAsync(index);
```

```
    Console.WriteLine(result);
}

static async Task BulkInsertAsync(SearchClient client)
{
    Console.WriteLine("Download data file");
    using HttpClient httpClient = new();

    var csv = await httpClient.GetStringAsync(BOOKS_URL);

    Console.WriteLine("Reading and parsing raw CSV data");
    var books =
        csv.ReplaceFirst("book_id", "id").FromCsv<List<BookModel>>();

    Console.WriteLine("Uploading bulk book data");
    _ = await client.UploadDocumentsAsync(books);

    Console.WriteLine("Finished bulk inserting book data");
}
```

2. Open an integrated terminal in Visual Studio Code for the project directory's subdirectory, `search-website-functions-v4/bulk-insert`, then run the following command to install the dependencies.

```bash
dotnet restore
```

Run the bulk import script for Search

1. Continue using the integrated terminal in Visual Studio for the project directory's subdirectory, `search-website-functions-v4/bulk-insert`, to run the following bash command to run the `Program.cs` script:

```bash
dotnet run
```

2. As the code runs, the console displays progress.

3. When the upload is complete, the last statement printed to the console is "Finished bulk inserting book data".

Review the new Search Index

Once the upload completes, the search index is ready to use. Review your new index in Azure portal.

1. In Azure portal, find the search service you created in the previous step.

2. On the Overview page, select the **Indexes** tab in the middle of the page, and then select the index you created in the previous step.

3. By default, the index opens in the **Search explorer** tab. Select **Search** to return documents from the index.

Rollback bulk import file changes

Use the following git command in the Visual Studio Code integrated terminal at the `bulk-insert` directory, to roll back the changes. They aren't needed to continue the tutorial and you shouldn't save or push these secrets to your repo.

```git
git checkout .
```

Copy your Search resource name

Note your **Search resource name**. You'll need this to connect the Azure Function app to your search resource.

> ⊗ **Caution**
>
> While you may be tempted to use your search admin key in the Azure Function, that isn't following the principle of least privilege. The Azure Function will use the query key to conform to least privilege.

Next steps

Deploy your Static Web App

Additional resources

🕮 Documentation

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure Cognitive Search index.

.NET tutorial: Search integration overview - Azure Cognitive Search

Technical overview and setup for adding search to a website and deploying to Azure Static Web App with .NET.

C# tutorial indexing multiple Azure data sources - Azure Cognitive Search

Learn how to import data from multiple data sources into a single Azure Cognitive Search index using indexers. This tutorial and sample code are in C#.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Add spell check to queries - Azure Cognitive Search

Attach spelling correction to the query pipeline, to fix typos on query terms before executing the query.

Text normalization for filters, facets, sort - Azure Cognitive Search

Specify normalizers to text fields in an index to customize the strict keyword matching behavior in filtering, faceting and sorting.

.NET tutorial: Search integration highlights - Azure Cognitive Search

Understand the .NET SDK Search integration queries used in the Search-enabled website with this cheat sheet.

Quickstart: Create a search index using REST APIs - Azure Cognitive Search

In this REST API quickstart, learn how to call the Azure Cognitive Search REST APIs using Postman.

Show 5 more

3 - Deploy the search-enabled .NET website

Article • 11/02/2022 • 6 minutes to read

Deploy the search-enabled website as an Azure Static Web Apps site. This deployment includes both the React app and the Function app.

The Static Web app pulls the information and files for deployment from GitHub using your fork of the samples repository.

Create a Static Web App in Visual Studio Code

1. Select **Azure** from the Activity Bar, then open **Resources** from the Side bar.

2. Right-click **Static Web Apps** and then select **Create Static Web App (Advanced)**.

3. If you see a pop-up window in VS Code asking which branch you want to deploy from, select the default branch, usually **master** or **main**.

This setting means only changes you commit to that branch are deployed to your static web app.

4. If you see a pop-up window asking you to commit your changes, don't do this. The secrets from the bulk import step shouldn't be committed to the repository.

   To roll back the changes, in VS Code select the Source Control icon in the Activity bar, then select each changed file in the Changes list and select the **Discard changes** icon.

5. Follow the prompts to provide the following information:

Prompt	Enter
Select a resource group for new resources.	Use the resource group you created for this tutorial.
Enter the name for the new Static Web App.	Create a unique name for your resource. For example, you can prepend your name to the repository name such as, `joansmith-azure-search-dotnet-samples`.
Select a SKU	Select the free SKU for this tutorial.
Select a location for new resources.	Select a region close to you.
Choose build preset to configure default project structure.	Select **Custom**
Select the location of your application code	`search-website-functions-v4/client` This is the path, from the root of the repository, to your static web app.
Enter the path of your build output...	`build` This is the path, from your static web app, to your generated files.

6. The resource is created and a notification window appears.

   When the resource is created, it creates a GitHub action file on GitHub.

7. Select **Open Actions in GitHub** from the Notifications. This opens a browser window pointed to your forked repo.

   Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

8. Pull the new GitHub action file to your local computer by synchronizing your local fork with your remote fork:

```bash
git pull origin main
```

   - *origin* refers to your forked repo.
   - *main* refers to the default branch.

9. In Visual Studio file explorer, find and open the workflow file in the `./.github/workflows/` directory. The file path and name looks *something* `.github\workflows\azure-static-web-apps-lemon-mushroom-0e1bd060f.yml`.

   The *part* of the YAML file relevant to the static web app is shown below:

```yml
name: Azure Static Web Apps CI/CD

on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened, closed]
    branches:
      - main

jobs:
  build_and_deploy_job:
    if: github.event_name == 'push' || (github.event_name ==
'pull_request' && github.event.action != 'closed')
    runs-on: ubuntu-latest
    name: Build and Deploy Job
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: true
      - name: Build And Deploy
        id: builddeploy
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          repo_token: ${{ secrets.GITHUB_TOKEN }} # Used for Github
integrations (i.e. PR comments)
          action: "upload"
```

```
          ###### Repository/Build Configurations - These values can be
configured to match your app requirements. ######
          # For more information regarding Static Web App workflow
configurations, please visit: https://aka.ms/swaworkflowconfig
          app_location: "search-website-functions-v4/client" # App
source code path
          api_location: "search-website-functions-v4/api" # Api source
code path - optional
          output_location: "build" # Built app content directory -
optional
          ###### End of Repository/Build Configurations ######
```

```yml
  close_pull_request_job:
    if: github.event_name == 'pull_request' && github.event.action ==
'closed'
    runs-on: ubuntu-latest
    name: Close Pull Request Job
    steps:
      - name: Close Pull Request
        id: closepullrequest
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          action: "close"
```

10. Edit your action file to contain the `api_location` property. If your local file doesn't have the property, add it below the `app_location` property.

```yml
api_location: "search-website-functions-v4/api"
```

11. Commit changes to your local repository.

```Bash
git add *.yml && git commit -m "update action for static web app"
```

12. Push changes to GitHub.

```Bash
git push origin main
```

The updated action in your remote fork creates a new build and deploy to your static web app. Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

Get Cognitive Search query key in Visual Studio Code

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the Side bar, select your Azure subscription under the **Azure: Cognitive Search** area, then right-click on your Search resource and select **Copy Query Key**.

3. Keep this query key, you'll need to use it in the next section. The query key is able to query your Index.

Add configuration settings in Azure portal

The Azure Function app won't return Search data until the Search secrets are in settings.

1. Select **Azure** from the Activity Bar.

2. Right-click on your Static Web Apps resource then select **Open in Portal**.

3. Select **Configuration** then select **+ Add**.

4. Add each of the following settings:

Setting	Your Search resource value
SearchApiKey	Your Search query key
SearchServiceName	Your Search resource name
SearchIndexName	`good-books`
SearchFacets	`authors*,language_code`

Azure Cognitive Search requires different syntax for filtering collections than it does for strings. Add a `*` after a field name to denote that the field is of type

`Collection(Edm.String)`. This allows the Azure Function to add filters correctly to queries.

5. Select **Save** to save the settings.

6. Return to VS Code.

7. Refresh your static web app to see the application settings.

Use search in your static web app

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Static Web Apps` area and find the static web app you created for this tutorial.
3. Right-click the static web app name and select **Browse site**.

4. Select **Open** in the pop-up dialog.

5. In the website search bar, enter a search query such as `code`, so the suggest feature suggests book titles. Select a suggestion or continue entering your own query. Press enter when you've completed your search query.

6. Review the results then select one of the books to see more details.

Troubleshooting

If the web app didn't deploy or work, use the following list to determine and fix the issue:

- **Did the deployment succeed?**

  In order to determine if your deployment succeeded, you need to go to *your* fork of the sample repo and review the success or failure of the GitHub action. There should be only 1 action and it should have static web app settings for the

`app_location`, `api_location`, and `output_location`. If the action didn't deploy successfully, dive into the action logs and look for the last failure.

- **Does the client (front-end) application work?**

  You should be able to get to your web app and it should successfully display. If the deployment succeeded but the website doesn't display, this may be an issue with how the static web app is configured for rebuilding the app, once it is on Azure.

- **Does the API (serverless back-end) application work?**

  You should be able to interact with the client app, searching for books and filtering. If the form doesn't return any values, open the browser's developer tools, and determine if the HTTP calls to the API were successful. If the calls weren't successful, the most likely reason if the static web app configurations for the API endpoint name and Search query key are incorrect.

Clean up resources

To clean up the resources created in this tutorial, delete the resource group.

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Resource Groups` area and find the resource group you created for this tutorial.
3. Right-click the resource group name then select **Delete**. This deletes both the Search and Static Web Apps resources.
4. If you no longer want the GitHub fork of the sample, remember to delete that on GitHub. Go to your fork's **Settings** then delete the fork.

Next steps

- Understand Search integration for the search-enabled website

4 - .NET Search integration cheat sheet

Article • 11/02/2022 • 9 minutes to read

In the previous lessons, you added search to a Static Web App. This lesson highlights the essential steps that establish integration. If you're looking for a cheat sheet on how to integrate search into your web app, this article explains what you need to know.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

Azure SDK Azure.Search.Documents

The Function app uses the Azure SDK for Cognitive Search:

- NuGet: Azure.Search.Documents
- Reference Documentation: Client Library

The Function app authenticates through the SDK to the cloud-based Cognitive Search API using your resource name, resource key, and index name. The secrets are stored in the Static Web App settings and pulled in to the Function as environment variables.

Configure secrets in a local.settings.json file

JSON

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "",
    "FUNCTIONS_WORKER_RUNTIME": "dotnet-isolated",
    "SearchApiKey": "",
    "SearchServiceName": "",
    "SearchIndexName": "good-books"
  },
  "Host": {
    "CORS": "*"
  }
}
```

Azure Function: Search the catalog

The `Search` API takes a search term and searches across the documents in the Search Index, returning a list of matches.

The Azure Function pulls in the Search configuration information, and fulfills the query.

C#

```
using Azure;
using Azure.Core.Serialization;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;
using System.Net;
using System.Text.Json;
using System.Text.Json.Serialization;
using WebSearch.Models;
using SearchFilter = WebSearch.Models.SearchFilter;

namespace WebSearch.Function
{
    public class Search
    {
        private static string searchApiKey =
Environment.GetEnvironmentVariable("SearchApiKey",
EnvironmentVariableTarget.Process);
        private static string searchServiceName =
Environment.GetEnvironmentVariable("SearchServiceName",
EnvironmentVariableTarget.Process);
        private static string searchIndexName =
Environment.GetEnvironmentVariable("SearchIndexName",
EnvironmentVariableTarget.Process) ?? "good-books";

        private readonly ILogger<Lookup> _logger;

        public Search(ILogger<Lookup> logger)
        {
            _logger = logger;
        }

        [Function("search")]
        public async Task<HttpResponseData> RunAsync(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req,
            FunctionContext executionContext)
        {
            string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
            var data = JsonSerializer.Deserialize<RequestBodySearch>
(requestBody);

            // Cognitive Search
```

```
            Uri serviceEndpoint =
new($"https://{searchServiceName}.search.windows.net/");

            SearchClient searchClient = new(
                serviceEndpoint,
                searchIndexName,
                new AzureKeyCredential(searchApiKey)
            );

            SearchOptions options = new()

            {
                Size = data.Size,
                Skip = data.Skip,
                IncludeTotalCount = true,
                Filter = CreateFilterExpression(data.Filters)
            };
            options.Facets.Add("authors");
            options.Facets.Add("language_code");

            SearchResults<SearchDocument> searchResults =
searchClient.Search<SearchDocument>(data.SearchText, options);

            var facetOutput = new Dictionary<string, IList<FacetValue>>();
            foreach (var facetResult in searchResults.Facets)
            {
                facetOutput[facetResult.Key] = facetResult.Value
                            .Select(x => new FacetValue { value =
x.Value.ToString(), count = x.Count })

                            .ToList();
            }

            // Data to return
            var output = new SearchOutput
            {
                Count = searchResults.TotalCount,
                Results = searchResults.GetResults().ToList(),
                Facets = facetOutput
            };

            var response = req.CreateResponse(HttpStatusCode.Found);

            // Serialize data
            var serializer = new JsonObjectSerializer(
                new JsonSerializerOptions(JsonSerializerDefaults.Web));
            await response.WriteAsJsonAsync(output, serializer);

            return response;
        }

        public static string CreateFilterExpression(List<SearchFilter>
filters)
        {
            if (filters is null or { Count: <= 0 })
```

```
            {
                return null;
            }

            List<string> filterExpressions = new();


            List<SearchFilter> authorFilters = filters.Where(f => f.field ==
"authors").ToList();
            List<SearchFilter> languageFilters = filters.Where(f => f.field
== "language_code").ToList();

            List<string> authorFilterValues = authorFilters.Select(f =>
f.value).ToList();

            if (authorFilterValues.Count > 0)
            {
                string filterStr = string.Join(",", authorFilterValues);
                filterExpressions.Add($"{"authors"}/any(t: search.in(t,
'{filterStr}', ','))");
            }

            List<string> languageFilterValues = languageFilters.Select(f =>
f.value).ToList();
            foreach (var value in languageFilterValues)
            {
                filterExpressions.Add($"language_code eq '{value}'");
            }

            return string.Join(" and ", filterExpressions);
        }
    }
}
```

Client: Search from the catalog

Call the Azure Function in the React client with the following code.

JavaScript

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import CircularProgress  from '@material-ui/core/CircularProgress';
import { useLocation, useNavigate } from "react-router-dom";

import Results from '../../components/Results/Results';
import Pager from '../../components/Pager/Pager';
import Facets from '../../components/Facets/Facets';
import SearchBar from '../../components/SearchBar/SearchBar';

import apiBaseUrl from "../../config";
import "./Search.css";
```

```
export default function Search() {

  let location = useLocation();
  const navigate = useNavigate();

  const [ results, setResults ] = useState([]);
  const [ resultCount, setResultCount ] = useState(0);
  const [ currentPage, setCurrentPage ] = useState(1);
  const [ q, setQ ] = useState(new URLSearchParams(location.search).get('q')
?? "*");
  const [ top ] = useState(new URLSearchParams(location.search).get('top')
?? 8);
  const [ skip, setSkip ] = useState(new
URLSearchParams(location.search).get('skip') ?? 0);
  const [ filters, setFilters ] = useState([]);
  const [ facets, setFacets ] = useState({});
  const [ isLoading, setIsLoading ] = useState(true);

  let resultsPerPage = top;

  useEffect(() => {
    setIsLoading(true);
    setSkip((currentPage-1) * top);
    const body = {
      q: q,
      top: top,
      skip: skip,
      filters: filters
    };

    axios.post( `${apiBaseUrl || ""}/api/search`, body)
      .then(response => {
            console.log(JSON.stringify(response.data))
            setResults(response.data.results);
            setFacets(response.data.facets);
            setResultCount(response.data.count);
            setIsLoading(false);
        } )
        .catch(error => {
            console.log(error);
            setIsLoading(false);
        });

  }, [q, top, skip, filters, currentPage]);

  // pushing the new search term to history when q is updated
  // allows the back button to work as expected when coming back from the
details page
  useEffect(() => {
    navigate('/search?q=' + q);
    setCurrentPage(1);
    setFilters([]);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [q]);
```

```
  let postSearchHandler = (searchTerm) => {
    //console.log(searchTerm);
    setQ(searchTerm);
  }

  var body;
  if (isLoading) {
    body = (
      <div className="col-md-9">
        <CircularProgress />
      </div>);
  } else {
    body = (
      <div className="col-md-9">
        <Results documents={results} top={top} skip={skip} count=
{resultCount}></Results>
        <Pager className="pager-style" currentPage={currentPage}
resultCount={resultCount} resultsPerPage={resultsPerPage} setCurrentPage=
{setCurrentPage}></Pager>
      </div>
    )
  }

  return (
    <main className="main main--search container-fluid">

      <div className="row">
        <div className="col-md-3">
          <div className="search-bar">
            <SearchBar postSearchHandler={postSearchHandler} q={q}>
</SearchBar>
          </div>
          <Facets facets={facets} filters={filters} setFilters={setFilters}>
</Facets>
        </div>
        {body}
      </div>
    </main>
  );
}
```

Azure Function: Suggestions from the catalog

The `Suggest` API takes a search term while a user is typing and suggests search terms such as book titles and authors across the documents in the search index, returning a small list of matches.

The search suggester, `sg`, is defined in the schema file used during bulk upload.

C#

```
using Azure;
using Azure.Core.Serialization;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
using Microsoft.Extensions.Logging;
using System.Net;
using System.Text.Json;
using WebSearch.Models;

namespace WebSearch.Function
{
    public class Suggest
    {
        private static string searchApiKey =
Environment.GetEnvironmentVariable("SearchApiKey",
EnvironmentVariableTarget.Process);
        private static string searchServiceName =
Environment.GetEnvironmentVariable("SearchServiceName",
EnvironmentVariableTarget.Process);
        private static string searchIndexName =
Environment.GetEnvironmentVariable("SearchIndexName",
EnvironmentVariableTarget.Process) ?? "good-books";

        private readonly ILogger<Lookup> _logger;

        public Suggest(ILogger<Lookup> logger)
        {
            _logger = logger;
        }

        [Function("suggest")]
        public async Task<HttpResponseData> RunAsync(
            [HttpTrigger(AuthorizationLevel.Anonymous, "post")]
HttpRequestData req,
            FunctionContext executionContext)
        {
            // Get Document Id
            string requestBody = await new
StreamReader(req.Body).ReadToEndAsync();
            var data = JsonSerializer.Deserialize<RequestBodySuggest>
(requestBody);

            // Cognitive Search
            Uri serviceEndpoint =
new($"https://{searchServiceName}.search.windows.net/");
```

```
            SearchClient searchClient = new(

                serviceEndpoint,
                searchIndexName,
                new AzureKeyCredential(searchApiKey)
            );

            SuggestOptions options = new()

            {
                Size = data.Size
            };

            var suggesterResponse = await
searchClient.SuggestAsync<BookModel>(data.SearchText, data.SuggesterName,
options);

            // Data to return
            var searchSuggestions = new Dictionary<string,
List<SearchSuggestion<BookModel>>>
            {
                ["suggestions"] = suggesterResponse.Value.Results.ToList()
            };

            var response = req.CreateResponse(HttpStatusCode.Found);

            // Serialize data
            var serializer = new JsonObjectSerializer(
                new JsonSerializerOptions(JsonSerializerDefaults.Web));
            await response.WriteAsJsonAsync(searchSuggestions, serializer);

            return response;
        }
    }
}
```

Client: Suggestions from the catalog

The Suggest function API is called in the React app at `\client\src\components\SearchBar\SearchBar.js` as part of component initialization:

JavaScript

```
import React, {useState, useEffect} from 'react';
import axios from 'axios';
import Suggestions from './Suggestions/Suggestions';

import "./SearchBar.css";

export default function SearchBar(props) {
```

```
    let [q, setQ] = useState("");
    let [suggestions, setSuggestions] = useState([]);
    let [showSuggestions, setShowSuggestions] = useState(false);

    const onSearchHandler = () => {
        props.postSearchHandler(q);
        setShowSuggestions(false);
    }

    const suggestionClickHandler = (s) => {
        document.getElementById("search-box").value = s;
        setShowSuggestions(false);
        props.postSearchHandler(s);
    }

    const onEnterButton = (event) => {
        if (event.keyCode === 13) {
            onSearchHandler();
        }
    }

    const onChangeHandler = () => {
        var searchTerm = document.getElementById("search-box").value;
        setShowSuggestions(true);
        setQ(searchTerm);

        // use this prop if you want to make the search more reactive
        if (props.searchChangeHandler) {
            props.searchChangeHandler(searchTerm);
        }
    }

    useEffect(_ =>{
        const timer = setTimeout(() => {
            const body = {
                q: q,
                top: 5,
                suggester: 'sg'
            };

            if (q === '') {
                setSuggestions([]);
            } else {
                axios.post( '/api/suggest', body)
                .then(response => {
                    console.log(JSON.stringify(response.data))
                    setSuggestions(response.data.suggestions);
                } )
                .catch(error => {
                    console.log(error);
                    setSuggestions([]);
                });
            }
        }, 300);
        return () => clearTimeout(timer);
```

```
    }, [q, props]);

    var suggestionDiv;
    if (showSuggestions) {
        suggestionDiv = (<Suggestions suggestions={suggestions}
suggestionClickHandler={(s) => suggestionClickHandler(s)}></Suggestions>);
    } else {
        suggestionDiv = (<div></div>);
    }

    return (
        <div >
            <div className="input-group" onKeyDown={e => onEnterButton(e)}>
                <div className="suggestions" >
                    <input
                        autoComplete="off" // setting for browsers; not the
app
                        type="text"
                        id="search-box"
                        className="form-control rounded-0"
                        placeholder="What are you looking for?"
                        onChange={onChangeHandler}
                        defaultValue={props.q}
                        onBlur={() => setShowSuggestions(false)}
                        onClick={() => setShowSuggestions(true)}>
                    </input>
                    {suggestionDiv}
                </div>
                <div className="input-group-btn">
                    <button className="btn btn-primary rounded-0"
type="submit" onClick={onSearchHandler}>
                        Search
                    </button>
                </div>
            </div>
        </div>
    );
};
```

Azure Function: Get specific document

The `Lookup` API takes an ID and returns the document object from the Search Index.

C#

```
using Azure;
using Azure.Core.Serialization;
using Azure.Search.Documents;
using Azure.Search.Documents.Models;
using Microsoft.Azure.Functions.Worker;
using Microsoft.Azure.Functions.Worker.Http;
```

```
using Microsoft.Extensions.Logging;
using System.Net;
using System.Text.Json;
using WebSearch.Models;

namespace WebSearch.Function
{
    public class Lookup
    {
        private static string searchApiKey =
Environment.GetEnvironmentVariable("SearchApiKey",
EnvironmentVariableTarget.Process);
        private static string searchServiceName =
Environment.GetEnvironmentVariable("SearchServiceName",
EnvironmentVariableTarget.Process);
        private static string searchIndexName =
Environment.GetEnvironmentVariable("SearchIndexName",
EnvironmentVariableTarget.Process) ?? "good-books";

        private readonly ILogger<Lookup> _logger;

        public Lookup(ILogger<Lookup> logger)
        {
            _logger = logger;
        }


        [Function("lookup")]
        public async Task<HttpResponseData> RunAsync(
            [HttpTrigger(AuthorizationLevel.Anonymous, "get", "post")]
HttpRequestData req,
            FunctionContext executionContext)
        {

            // Get Document Id
            var query =
System.Web.HttpUtility.ParseQueryString(req.Url.Query);
            string documentId = query["id"].ToString();

            // Cognitive Search
            Uri serviceEndpoint =
new($"https://{searchServiceName}.search.windows.net/");

            SearchClient searchClient = new(

                serviceEndpoint,
                searchIndexName,
                new AzureKeyCredential(searchApiKey)
            );

            var getDocumentResponse = await
searchClient.GetDocumentAsync<SearchDocument>(documentId);

            // Data to return
            var output = new LookupOutput
```

```
            {
                Document = getDocumentResponse.Value
            };

            var response = req.CreateResponse(HttpStatusCode.Found);
            response.Headers.Add("Content-Type", "application/json;
charset=utf-8");

            // Serialize data
            var serializer = new JsonObjectSerializer(
                new JsonSerializerOptions(JsonSerializerDefaults.Web));
            await response.WriteAsJsonAsync(output, serializer);

            return response;
        }
    }
}
```

Client: Get specific document

This function API is called in the React app at `\client\src\pages\Details\Detail.js` as part of component initialization:

JavaScript

```
import React, { useState, useEffect } from "react";
import { useParams } from 'react-router-dom';
import Rating from '@material-ui/lab/Rating';
import CircularProgress from '@material-ui/core/CircularProgress';
import axios from 'axios';
import apiBaseUrl from "../../config";

import "./Details.css";

export default function Details() {

  let { id } = useParams();
  const [document, setDocument] = useState({});
  const [selectedTab, setTab] = useState(0);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setIsLoading(true);
    // console.log(id);
```

```
    axios.get(`${apiBaseUrl || ""}/api/lookup?id=${id}`)
      .then(response => {
        console.log(JSON.stringify(response.data))
        const doc = response.data.document;
        setDocument(doc);
        setIsLoading(false);
      })
      .catch(error => {
        console.log(error);
        setIsLoading(false);
      });
```

```
  }, [id]);

  // View default is loading with no active tab
  let detailsBody = (<CircularProgress />),
      resultStyle = "nav-link",
      rawStyle    = "nav-link";

  if (!isLoading && document) {
    // View result
    if (selectedTab === 0) {
      resultStyle += " active";
      detailsBody = (
        <div className="card-body">
          <h5 className="card-title">{document.original_title}</h5>
          <img className="image" src={document.image_url} alt="Book cover">
</img>
          <p className="card-text">{document.authors?.join('; ')} -
{document.original_publication_year}</p>
          <p className="card-text">ISBN {document.isbn}</p>
          <Rating name="half-rating-read" value=
{parseInt(document.average_rating)} precision={0.1} readOnly></Rating>
          <p className="card-text">{document.ratings_count} Ratings</p>
        </div>
      );
    }

    // View raw data
    else {
      rawStyle += " active";
      detailsBody = (
        <div className="card-body text-left">
          <pre><code>
            {JSON.stringify(document, null, 2)}
          </code></pre>
        </div>
      );
    }
  }

  return (
    <main className="main main--details container fluid">
      <div className="card text-center result-container">
        <div className="card-header">
```

```
          <ul className="nav nav-tabs card-header-tabs">
              <li className="nav-item"><button className={resultStyle}
onClick={() => setTab(0)}>Result</button></li>
              <li className="nav-item"><button className={rawStyle} onClick=
{() => setTab(1)}>Raw Data</button></li>
          </ul>
        </div>
        {detailsBody}
      </div>
    </main>
  );
}
```

C# models to support function app

The following models are used to support the functions in this app.

C#

```
using Azure.Search.Documents.Models;
using System.Text.Json.Serialization;

namespace WebSearch.Models
{
    public class RequestBodyLookUp
    {
        [JsonPropertyName("id")]
        public string Id { get; set; }
    }

    public class RequestBodySuggest
    {
        [JsonPropertyName("q")]
        public string SearchText { get; set; }

        [JsonPropertyName("top")]
        public int Size { get; set; }

        [JsonPropertyName("suggester")]
        public string SuggesterName { get; set; }
    }

    public class RequestBodySearch
    {
        [JsonPropertyName("q")]
        public string SearchText { get; set; }

        [JsonPropertyName("skip")]
        public int Skip { get; set; }

        [JsonPropertyName("top")]
        public int Size { get; set; }
```

```
        [JsonPropertyName("filters")]
        public List<SearchFilter> Filters { get; set; }
    }

    public class SearchFilter
    {
        public string field { get; set; }
        public string value { get; set; }
    }

    public class FacetValue
    {
        public string value { get; set; }
        public long? count { get; set; }
    }

    class SearchOutput
    {
        [JsonPropertyName("count")]
        public long? Count { get; set; }
        [JsonPropertyName("results")]
        public List<SearchResult<SearchDocument>> Results { get; set; }
        [JsonPropertyName("facets")]
        public Dictionary<String, IList<FacetValue>> Facets { get; set; }
    }
    class LookupOutput
    {
        [JsonPropertyName("document")]
        public SearchDocument Document { get; set; }
    }
    public class BookModel
    {
        public string id { get; set; }
        public decimal? goodreads_book_id { get; set; }
        public decimal? best_book_id { get; set; }
        public decimal? work_id { get; set; }
        public decimal? books_count { get; set; }
        public string isbn { get; set; }
        public string isbn13 { get; set; }
        public string[] authors { get; set; }
        public decimal? original_publication_year { get; set; }
        public string original_title { get; set; }
        public string title { get; set; }
        public string language_code { get; set; }
        public double? average_rating { get; set; }
        public decimal? ratings_count { get; set; }
        public decimal? work_ratings_count { get; set; }
        public decimal? work_text_reviews_count { get; set; }
        public decimal? ratings_1 { get; set; }
        public decimal? ratings_2 { get; set; }
        public decimal? ratings_3 { get; set; }
        public decimal? ratings_4 { get; set; }
        public decimal? ratings_5 { get; set; }
        public string image_url { get; set; }
```

```
        public string small_image_url { get; set; }
    }
}
```

Next steps

- Index Azure SQL data

Additional resources

Documentation

.NET tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with .NET.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Create or Update Index (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Create or Update Index REST API for Azure Cognitive Search.

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure Cognitive Search index.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Azure Cognitive Search client library for .NET - Azure for .NET Developers

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Show 5 more

Training

Learning path

Microsoft Search fundamentals - Training

Learn about Microsoft Search including where users can search, the answers and results they'll see,

and how you can tailor the search experience for your organization.

1 - Overview of adding search to a website

Article • 11/23/2022 • 2 minutes to read

This tutorial builds a website to search through a catalog of books then deploys the website to an Azure Static Web Apps resource.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

What does the sample do?

This sample website provides access to a catalog of 10,000 books. A user can search the catalog by entering text in the search bar. While the user enters text, the website uses the search index's suggest feature to complete the text. Once the query finishes, the list of books is displayed with a portion of the details. A user can select a book to see all the details, stored in the search index, of the book.

The search experience includes:

- Search – provides search functionality for the application.
- Suggest – provides suggestions as the user is typing in the search bar.
- Document Lookup – looks up a document by ID to retrieve all of its contents for the details page.

How is the sample organized?

The sample includes the following:

App	Purpose	GitHub Repository Location
Client	React app (presentation layer) to display books, with search. It calls the Azure Function app.	/search-website-functions-v4/client
Server	Azure Function app (business layer) - calls the Azure Cognitive Search API using JavaScript SDK	/search-website-functions-v4/api
Bulk insert	JavaScript file to create the index and add documents to it.	/search-website-functions-v4/bulk-insert

Set up your development environment

Install the following for your local development environment.

- Node.js LTS
  - Select latest runtime and version from this list of supported language versions.
  - If you have a different version of Node.js installed on your local computer, consider using Node Version Manager (nvm) or a Docker container.
- Git
- Visual Studio Code and the following extensions
  - Azure Static Web App
- Optional:
  - This tutorial doesn't run the Azure Function API locally. If you intend to run it locally, you need to install azure-functions-core-tools globally with the following bash command:

```bash
npm install -g azure-functions-core-tools@4
```

Fork and clone the search sample with git

Forking the sample repository is critical to be able to deploy the Static Web App. The static web app determines the build actions and deployment content based on your

own GitHub fork location. Code execution in the Static Web App is remote, with the static web app reading from the code in your forked sample.

1. On GitHub, fork the sample repository.

   Complete the fork process in your web browser with your GitHub account. This tutorial uses your fork as part of the deployment to an Azure Static Web App.

1. At a Bash terminal, download your forked sample application to your local computer.

   Replace `YOUR-GITHUB-ALIAS` with your GitHub alias.

   ```bash
   git clone https://github.com/YOUR-GITHUB-ALIAS/azure-search-javascript-samples
   ```

2. At the same Bash terminal, go into your forked repository for this website search example:

   ```bash
   cd azure-search-javascript-samples
   ```

3. Use the Visual Studio Code command, `code .` to open your forked repository. The remaining tasks are accomplished from Visual Studio Code, unless specified.

   ```bash
   code .
   ```

Create a resource group for your Azure resources

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the **Resources** section, select Add (+), and then select **Create Resource Group**.

3. Enter a resource group name, such as `cognitive-search-website-tutorial`.

4. Select a location close to you.

5. When you create the Cognitive Search and Static Web Apps resources, later in the tutorial, use this resource group.

   Creating a resource group gives you a logical unit to manage the resources, including deleting them when you're finished using them.

Next steps

- Create a Search Index and load with documents
- Deploy your Static Web App

Additional resources

🕮 Documentation

JavaScript tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with JavaScript using the npm SDK @azure/search-documents.

Azure Cognitive Search client library for JavaScript

Python tutorial: Deploy search-enabled website - Azure Cognitive Search

Deploy search-enabled Python website to Azure Static web app.

.NET tutorial: Search integration overview - Azure Cognitive Search

Technical overview and setup for adding search to a website and deploying to Azure Static Web App with .NET.

Quickstart: Create a search index in JavaScript - Azure Cognitive Search

In this JavaScript quickstart, learn how to create an index, load data, and run queries on Azure Cognitive Search using JavaScript

.NET tutorial: Deploy search-enabled website - Azure Cognitive Search

Deploy search-enabled website with .NET apis to Azure Static web app.

JavaScript samples - Azure Cognitive Search

Find Azure Cognitive Search demo JavaScript code samples that use the Azure .NET SDK for JavaScript.

JavaScript tutorial: Search integration highlights - Azure Cognitive Search

Understand the JavaScript SDK Search integration queries used in the Search-enabled website with this cheat sheet.

Show 5 more

Training

Learning path

Azure Static Web Apps - Training

Azure Static Web Apps is a modern web app service that offers streamlined full-stack development from source code to global high availability.

2 - Create and load Search Index with JavaScript

Article • 01/23/2023 • 5 minutes to read

Continue to build your search-enabled website by following these steps:

- Create a search resource
- Create a new index
- Import data with JavaScript using the sample script and Azure SDK @azure/search-documents.

Create an Azure Cognitive Search resource

Create a new search resource using PowerShell and the **Az.Search** module. In this section, you'll also create a query key used for read-access to the index, and get the built-in admin key used for adding objects.

1. In Visual Studio Code, open a new terminal window.

2. Connect to Azure:

   PowerShell

   ```
   Connect-AzAccount -TenantID <your-tenant-ID>
   ```

   > **Note**
   >
   > You might need to provide a tenant ID, which you can find in the Azure portal in **Portal settings** > **Directories + subscriptions**.

3. Before creating a new search service, you can list existing search services for your subscription to see if there's one you want to use:

   PowerShell

   ```
   Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
   ```

4. Load the **Az.Search** module:

   PowerShell

```
Install-Module -Name Az.Search
```

5. Create a new search service. Use the following cmdlet as a template, substituting valid values for the resource group, service name, tier, region, partitions, and replicas:

PowerShell

```
New-AzSearchService -ResourceGroupName "my resource group"  -Name
"myDemoSearchSvc" -Sku "Free" -Location "West US" -PartitionCount 1 -
ReplicaCount 1 -HostingMode Default
```

Prompt	Enter
Enter a globally unique name for the new search service.	**Remember this name**. This resource name becomes part of your resource endpoint.
Select a resource group for new resources	Use the resource group you created for this tutorial.
Select the SKU for your search service.	Use **Free** for this tutorial. You can't change a SKU pricing tier after the service is created.
Select a location for new resources.	Select a region close to you.

6. Create a query key that grants read access to a search service. Query keys have to be explicitly created. Copy the query key to Notepad so that you can paste it into the client code in a later step:

PowerShell

```
New-AzSearchQueryKey -ResourceGroupName "my resource group"  -
ServiceName "myDemoSearchSvc" -Name "mySrchQueryKey"
```

7. Get the search service admin API key that was automatically created for your search service. An admin API key provides write access to the search service. Copy either one of the admin keys to Notepad so that you can use it in the bulk import step that creates and loads an index:

PowerShell

```
Get-AzSearchAdminKeyPair  -ResourceGroupName "my resource group" -
ServiceName "myDemoSearchSvc"
```

Prepare the bulk import script for Search

The script uses the Azure SDK for Cognitive Search:

- npm package @azure/search-documents
- Reference Documentation

1. In Visual Studio Code, open the `bulk_insert_books.js` file in the subdirectory, `search-website-functions-v4/bulk-insert`, replace the following variables with your own values to authenticate with the Azure Search SDK:

   - YOUR-SEARCH-RESOURCE-NAME
   - YOUR-SEARCH-ADMIN-KEY

   JavaScript

```javascript
/**
 * Add documents into Search Index
 * Script creates a new index called `good-books` in your Search
resource.
 *
 * 1. Edit the values for your own Search resource.
 *     * YOUR-SEARCH-RESOURCE-NAME
 *     * YOUR-SEARCH-ADMIN-KEY
 * 2. Run script with `npm install && npm start`
 *
 */

const axios = require('axios');
const Papa = require('papaparse')
const { SearchClient, SearchIndexClient, AzureKeyCredential } =
require("@azure/search-documents");

const SEARCH_ENDPOINT = "https://YOUR-SEARCH-RESOURCE-
NAME.search.windows.net";
const SEARCH_KEY = "YOUR-SEARCH-ADMIN-KEY";

const SEARCH_INDEX_NAME = "good-books";
const SEARCH_INDEX_SCHEMA = require("./good-books-index.json");

const BOOKS_URL =
"https://raw.githubusercontent.com/zygmuntz/goodbooks-
10k/master/books.csv";
const BATCH_SIZE = 1000;

// Create Search service client
// used to upload docs into Index
```

```
const client = new SearchClient(
    SEARCH_ENDPOINT,
    SEARCH_INDEX_NAME,
    new AzureKeyCredential(SEARCH_KEY)
);

// Create Search service Index client
// used to create new Index
const clientIndex = new SearchIndexClient(
    SEARCH_ENDPOINT,
    new AzureKeyCredential(SEARCH_KEY)
);
```

```
// Insert docs into Search Index
// in batch
const insertData = async (data) => {

    let batch = 0;
    let batchArray = [];

    for (let i = 0; i < data.length; i++) {

        const row = data[i];

        // Convert string data to typed data
        // Types are defined in schema
        batchArray.push({
            "id": row.book_id,
            "goodreads_book_id": parseInt(row.goodreads_book_id),
            "best_book_id": parseInt(row.best_book_id),
            "work_id": parseInt(row.work_id),
            "books_count": !row.books_count ? 0 :
parseInt(row.books_count),
            "isbn": row.isbn,
            "isbn13": row.isbn13,
            "authors": row.authors.split(",").map(name => name.trim()),
            "original_publication_year": !row.original_publication_year
? 0 : parseInt(row.original_publication_year),
            "original_title": row.original_title,
            "title": row.title,
            "language_code": row.language_code,
            "average_rating": !row.average_rating ? 0 :
parseFloat(row.average_rating),
            "ratings_count": !row.ratings_count ? 0 :
parseInt(row.ratings_count),
            "work_ratings_count": !row.work_ratings_count ? 0 :
parseInt(row.work_ratings_count),
            "work_text_reviews_count": !row.work_text_reviews_count ? 0
: parseInt(row.work_text_reviews_count),
            "ratings_1": !row.ratings_1 ? 0 : parseInt(row.ratings_1),
            "ratings_2": !row.ratings_2 ? 0 : parseInt(row.ratings_2),
            "ratings_3": !row.ratings_3 ? 0 : parseInt(row.ratings_3),
            "ratings_4": !row.ratings_4 ? 0 : parseInt(row.ratings_4),
            "ratings_5": !row.ratings_5 ? 0 : parseInt(row.ratings_5),
            "image_url": row.image_url,
```

```
            "small_image_url": row.small_image_url
        })

        console.log(`${i}`);

        // Insert batch into Index
        if ((batchArray.length % BATCH_SIZE) === 0){
            await client.uploadDocuments(batchArray);

            console.log(`BATCH SENT`);
            batchArray = [];
        }

    }
    // Insert any final batch into Index
    if (batchArray.length > 0 ){
        await client.uploadDocuments(batchArray);

        console.log(`FINAL BATCH SENT`);
        batchArray = [];
    }
}
const bulkInsert = async () => {

    // Download CSV Data file
    const response = await axios.get(BOOKS_URL);
    const fileData = response.data;

    // convert CSV to JSON
    const dataObj = Papa.parse(fileData, {
        header: true,
        encoding: 'utf8',
        skipEmptyLines: true,
    })

    // Insert JSON into Search Index
    await insertData(dataObj.data)
}

// Create Search Index
async function createIndex() {

    SEARCH_INDEX_SCHEMA.name = SEARCH_INDEX_NAME;
    const result = await clientIndex.createIndex(SEARCH_INDEX_SCHEMA);

    console.log(result);
}

const main = async () => {

    await createIndex();
    console.log("index created");

    await bulkInsert();
}
```

```
main()
.then(() => console.log('done'))
.catch((err) => {
    console.log(`done +  failed ${err}`)
});
```

2. Open an integrated terminal in Visual Studio for the project directory's subdirectory, `search-website-functions-v4/bulk-insert`, and run the following command to install the dependencies.

```bash
npm install
```

Run the bulk import script for Search

1. Continue using the integrated terminal in Visual Studio for the project directory's subdirectory, `search-website-functions-v4/bulk-insert`, to run the following bash command to run the `bulk_insert_books.js` script:

```javascript
npm start
```

2. As the code runs, the console displays progress.

3. When the upload is complete, the last statement printed to the console is "done".

Review the new Search Index

Once the upload completes, the search index is ready to use. Review your new index in Azure portal.

1. In Azure portal, find the search service you created in the previous step.

2. On the Overview page, select the **Indexes** tab in the middle of the page, and then select the index you created in the previous step.

3. By default, the index opens in the **Search explorer** tab. Select **Search** to return documents from the index.

Rollback bulk import file changes

Use the following git command in the Visual Studio Code integrated terminal at the `bulk-insert` directory, to roll back the changes. They aren't needed to continue the tutorial and you shouldn't save or push these secrets to your repo.

```
git
```

```
git checkout .
```

Copy your Search resource name

Note your **Search resource name**. You'll need this to connect the Azure Function app to your search resource.

> ⊗ **Caution**
>
> While you may be tempted to use your search admin key in the Azure Function, that isn't following the principle of least privilege. The Azure Function will use the query key to conform to least privilege.

Next steps

Deploy your Static Web App

Additional resources

🕮 Documentation

Azure Cognitive Search client library for JavaScript

JavaScript tutorial: Search integration highlights - Azure Cognitive Search

Understand the JavaScript SDK Search integration queries used in the Search-enabled website with this cheat sheet.

.NET tutorial: Search integration overview - Azure Cognitive Search

Technical overview and setup for adding search to a website and deploying to Azure Static Web App with .NET.

Quickstart: Create a search index in JavaScript - Azure Cognitive Search

In this JavaScript quickstart, learn how to create an index, load data, and run queries on Azure Cognitive Search using JavaScript

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom,

predefined or language-specific alternatives.

.NET tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with .NET.

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure Cognitive Search index.

Show 5 more

3 - Deploy the search-enabled website

Article • 11/01/2022 • 6 minutes to read

Deploy the search-enabled website as an Azure Static Web Apps site. This deployment includes both the React app and the Function app.

The Static Web app pulls the information and files for deployment from GitHub using your fork of the samples repository.

Create a Static Web App in Visual Studio Code

1. Select **Azure** from the Activity Bar, then open **Resources** from the Side bar.

2. Right-click **Static Web Apps** and then select **Create Static Web App (Advanced)**.

3. If you see a pop-up window in VS Code asking which branch you want to deploy from, select the default branch, usually **master** or **main**.

This setting means only changes you commit to that branch are deployed to your static web app.

4. If you see a pop-up window asking you to commit your changes, don't do this. The secrets from the bulk import step shouldn't be committed to the repository.

   To roll back the changes, in VS Code select the Source Control icon in the Activity bar, then select each changed file in the Changes list and select the **Discard changes** icon.

5. Follow the prompts to provide the following information:

Prompt	Enter
Select a resource group for new resources.	Use the resource group you created for this tutorial.
Enter the name for the new Static Web App.	Create a unique name for your resource. For example, you can prepend your name to the repository name such as, `joansmith-azure-search-dotnet-samples`.
Select a SKU	Select the free SKU for this tutorial.
Select a location for new resources.	Select a region close to you.
Choose build preset to configure default project structure.	Select **Custom**
Select the location of your application code	`search-website-functions-v4/client` This is the path, from the root of the repository, to your static web app.
Enter the path of your build output...	`build` This is the path, from your static web app, to your generated files.

6. The resource is created and a notification window appears.

   When the resource is created, it creates a GitHub action file on GitHub.

7. Select **Open Actions in GitHub** from the Notifications. This opens a browser window pointed to your forked repo.

   Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

8. Pull the new GitHub action file to your local computer by synchronizing your local fork with your remote fork:

```bash
git pull origin main
```

   - *origin* refers to your forked repo.
   - *main* refers to the default branch.

9. In Visual Studio file explorer, find and open the workflow file in the `./.github/workflows/` directory. The file path and name looks *something* `.github\workflows\azure-static-web-apps-lemon-mushroom-0e1bd060f.yml`.

   The *part* of the YAML file relevant to the static web app is shown below:

```yml
name: Azure Static Web Apps CI/CD

on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened, closed]
    branches:
      - main

jobs:
  build_and_deploy_job:
    if: github.event_name == 'push' || (github.event_name ==
'pull_request' && github.event.action != 'closed')
    runs-on: ubuntu-latest
    name: Build and Deploy Job
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: true
      - name: Build And Deploy
        id: builddeploy
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          repo_token: ${{ secrets.GITHUB_TOKEN }} # Used for Github
integrations (i.e. PR comments)
          action: "upload"
```

```
          ###### Repository/Build Configurations - These values can be
configured to match your app requirements. ######
          # For more information regarding Static Web App workflow
configurations, please visit: https://aka.ms/swaworkflowconfig
          app_location: "search-website-functions-v4/client" # App
source code path
          api_location: "search-website-functions-v4/api" # Api source
code path - optional
          output_location: "build" # Built app content directory -
optional
          ###### End of Repository/Build Configurations ######
```

```yml
  close_pull_request_job:
    if: github.event_name == 'pull_request' && github.event.action ==
'closed'
    runs-on: ubuntu-latest
    name: Close Pull Request Job
    steps:
      - name: Close Pull Request
        id: closepullrequest
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          action: "close"
```

10. Edit your action file to contain the `api_location` property. If your local file doesn't have the property, add it below the `app_location` property.

```yml
api_location: "search-website-functions-v4/api"
```

11. Commit changes to your local repository.

```Bash
git add *.yml && git commit -m "update action for static web app"
```

12. Push changes to GitHub.

```Bash
git push origin main
```

The updated action in your remote fork creates a new build and deploy to your static web app. Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

Get Cognitive Search query key in Visual Studio Code

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the Side bar, select your Azure subscription under the **Azure: Cognitive Search** area, then right-click on your Search resource and select **Copy Query Key**.

3. Keep this query key, you'll need to use it in the next section. The query key is able to query your Index.

Add configuration settings in Azure portal

The Azure Function app won't return Search data until the Search secrets are in settings.

1. Select **Azure** from the Activity Bar.

2. Right-click on your Static Web Apps resource then select **Open in Portal**.

3. Select **Configuration** then select **+ Add**.

4. Add each of the following settings:

Setting	Your Search resource value
SearchApiKey	Your Search query key
SearchServiceName	Your Search resource name
SearchIndexName	`good-books`
SearchFacets	`authors*,language_code`

Azure Cognitive Search requires different syntax for filtering collections than it does for strings. Add a `*` after a field name to denote that the field is of type

`Collection(Edm.String)`. This allows the Azure Function to add filters correctly to queries.

5. Select **Save** to save the settings.

6. Return to VS Code.

7. Refresh your static web app to see the application settings.

Use search in your static web app

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Static Web Apps` area and find the static web app you created for this tutorial.
3. Right-click the static web app name and select **Browse site**.

4. Select **Open** in the pop-up dialog.

5. In the website search bar, enter a search query such as `code`, so the suggest feature suggests book titles. Select a suggestion or continue entering your own query. Press enter when you've completed your search query.

6. Review the results then select one of the books to see more details.

Troubleshooting

If the web app didn't deploy or work, use the following list to determine and fix the issue:

- **Did the deployment succeed?**

  In order to determine if your deployment succeeded, you need to go to *your* fork of the sample repo and review the success or failure of the GitHub action. There should be only 1 action and it should have static web app settings for the

`app_location`, `api_location`, and `output_location`. If the action didn't deploy successfully, dive into the action logs and look for the last failure.

- **Does the client (front-end) application work?**

  You should be able to get to your web app and it should successfully display. If the deployment succeeded but the website doesn't display, this may be an issue with how the static web app is configured for rebuilding the app, once it is on Azure.

- **Does the API (serverless back-end) application work?**

  You should be able to interact with the client app, searching for books and filtering. If the form doesn't return any values, open the browser's developer tools, and determine if the HTTP calls to the API were successful. If the calls weren't successful, the most likely reason if the static web app configurations for the API endpoint name and Search query key are incorrect.

Clean up resources

To clean up the resources created in this tutorial, delete the resource group.

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Resource Groups` area and find the resource group you created for this tutorial.
3. Right-click the resource group name then select **Delete**. This deletes both the Search and Static Web Apps resources.
4. If you no longer want the GitHub fork of the sample, remember to delete that on GitHub. Go to your fork's **Settings** then delete the fork.

Next steps

- Understand Search integration for the search-enabled website

---

Additional resources

🕮 Documentation

Changed and deleted blobs - Azure Cognitive Search

Indexers that index from Azure Storage can pick up new and changed content automatically. This

article describes the strategies.

Document operations using Azure Cognitive Search REST APIs

Learn which REST API calls are used to populate and query an Azure Cognitive Search index.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

moreLikeThis (preview) query feature - Azure Cognitive Search

Describes the moreLikeThis (preview) feature, which is available in preview versions of the Azure Cognitive Search REST API.

Analyze Text (2021-04-30-Preview Rest API)

Preview version of the Analyze Text REST API for Azure Cognitive Search.

JavaScript tutorial: Search integration highlights - Azure Cognitive Search

Understand the JavaScript SDK Search integration queries used in the Search-enabled website with this cheat sheet.

.NET tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with .NET.

Show 5 more

Training

Learning path

Azure Static Web Apps - Training

Azure Static Web Apps is a modern web app service that offers streamlined full-stack development from source code to global high availability.

4 - JavaScript Search integration cheat sheet

Article • 11/23/2022 • 8 minutes to read

In the previous lessons, you added search to a Static Web App. This lesson highlights the essential steps that establish integration. If you're looking for a cheat sheet on how to integrate search into your JavaScript app, this article explains what you need to know.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

Azure SDK @azure/search-documents

The Function app uses the Azure SDK for Cognitive Search:

- NPM: @azure/search-documents
- Reference Documentation: Client Library

The Function app authenticates through the SDK to the cloud-based Cognitive Search API using your resource name, resource key, and index name. The secrets are stored in the Static Web App settings and pulled in to the Function as environment variables.

Configure secrets in a configuration file

JavaScript

```javascript
const CONFIG = {
    SearchIndexName: process.env["SearchIndexName"] || "good-books",
    SearchApiKey: process.env["SearchApiKey"] || "",
    SearchServiceName: process.env["SearchServiceName"] || "",
    SearchFacets: process.env["SearchFacets"] || "authors*,language_code",
}
console.log(CONFIG);
if (!CONFIG.SearchIndexName || !CONFIG.SearchApiKey ||
!CONFIG.SearchServiceName) throw Error("./config.js::Cognitive Services key
is missing");

module.exports = { CONFIG };
```

Azure Function: Search the catalog

The `Search` API takes a search term and searches across the documents in the Search Index, returning a list of matches.

Routing for the Search API is contained in the function.json bindings.

The Azure Function pulls in the Search configuration information, and fulfills the query.

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");
const { CONFIG } = require("../config");

// Create a SearchClient to send queries
const client = new SearchClient(
    `https://` + CONFIG.SearchServiceName + `.search.windows.net/`,
    CONFIG.SearchIndexName,
    new AzureKeyCredential(CONFIG.SearchApiKey)
);

// creates filters in odata syntax
const createFilterExpression = (filterList, facets) => {
    let i = 0;
    let filterExpressions = [];

    while (i < filterList.length) {
        let field = filterList[i].field;
        let value = filterList[i].value;

        if (facets[field] === 'array') {
            filterExpressions.push(`${field}/any(t: search.in(t, '${value}',
','))`);
        } else {
            filterExpressions.push(`${field} eq '${value}'`);
        }
        i += 1;
    }

    return filterExpressions.join(' and ');
}

// reads in facets and gets type
// array facets should include a * at the end
// this is used to properly create filters
const readFacets = (facetString) => {
    let facets = facetString.split(",");
    let output = {};
    facets.forEach(function (f) {
        if (f.indexOf('*') > -1) {
            output[f.replace('*', '')] = 'array';
```

```
        } else {
            output[f] = 'string';
        }
    })

    return output;
}

module.exports = async function (context, req) {

    try {

        // Reading inputs from HTTP Request
        let q = (req.query.q || (req.body && req.body.q));
        const top = (req.query.top || (req.body && req.body.top));
        const skip = (req.query.skip || (req.body && req.body.skip));
        const filters = (req.query.filters || (req.body &&
req.body.filters));
        const facets = readFacets(CONFIG.SearchFacets);


        // If search term is empty, search everything
        if (!q || q === "") {
            q = "*";
        }

        // Creating SearchOptions for query
        let searchOptions = {
            top: top,
            skip: skip,
            includeTotalCount: true,
            facets: Object.keys(facets),
            filter: createFilterExpression(filters, facets)
        };

        // Sending the search request
        const searchResults = await client.search(q, searchOptions);

        // Getting results for output
        const output = [];
        for await (const result of searchResults.results) {
            output.push(result);
        }

        // Logging search results
        context.log(searchResults.count);

        // Creating the HTTP Response
        context.res = {
            // status: 200, /* Defaults to 200 */
            headers: {
                "Content-type": "application/json"
            },
            body: {
                count: searchResults.count,
```

```
                results: output,
                facets: searchResults.facets
            }
        };
    } catch (error) {
        context.log.error(error);

        // Creating the HTTP Response
        context.res = {
            status: 400,
            body: {
                innerStatusCode: error.statusCode || error.code,
                error: error.details || error.message
            }
        };
    }
};
```

Client: Search from the catalog

Call the Azure Function in the React client with the following code.

JavaScript

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import CircularProgress  from '@material-ui/core/CircularProgress';
import { useLocation, useNavigate } from "react-router-dom";

import Results from '../../components/Results/Results';
import Pager from '../../components/Pager/Pager';
import Facets from '../../components/Facets/Facets';
import SearchBar from '../../components/SearchBar/SearchBar';

import apiBaseUrl from "../../config";
import "./Search.css";

export default function Search() {

  let location = useLocation();
  const navigate = useNavigate();

  const [ results, setResults ] = useState([]);
  const [ resultCount, setResultCount ] = useState(0);
  const [ currentPage, setCurrentPage ] = useState(1);
  const [ q, setQ ] = useState(new URLSearchParams(location.search).get('q')
?? "*");
  const [ top ] = useState(new URLSearchParams(location.search).get('top')
?? 8);
  const [ skip, setSkip ] = useState(new
URLSearchParams(location.search).get('skip') ?? 0);
  const [ filters, setFilters ] = useState([]);
```

```
  const [ facets, setFacets ] = useState({});
  const [ isLoading, setIsLoading ] = useState(true);

  let resultsPerPage = top;

  useEffect(() => {
    setIsLoading(true);
    setSkip((currentPage-1) * top);
    const body = {
      q: q,
      top: top,
      skip: skip,
      filters: filters
    };

    axios.post( `${apiBaseUrl || ""}/api/search`, body)
      .then(response => {
            console.log(JSON.stringify(response.data))
            setResults(response.data.results);
            setFacets(response.data.facets);
            setResultCount(response.data.count);
            setIsLoading(false);
        } )
        .catch(error => {
            console.log(error);
            setIsLoading(false);
        });

  }, [q, top, skip, filters, currentPage]);

  // pushing the new search term to history when q is updated
  // allows the back button to work as expected when coming back from the
details page
  useEffect(() => {
    navigate('/search?q=' + q);
    setCurrentPage(1);
    setFilters([]);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [q]);


  let postSearchHandler = (searchTerm) => {
    //console.log(searchTerm);
    setQ(searchTerm);
  }

  var body;
  if (isLoading) {
    body = (
      <div className="col-md-9">
        <CircularProgress />
      </div>);
  } else {
    body = (
      <div className="col-md-9">
```

```
        <Results documents={results} top={top} skip={skip} count=
{resultCount}></Results>
        <Pager className="pager-style" currentPage={currentPage}
resultCount={resultCount} resultsPerPage={resultsPerPage} setCurrentPage=
{setCurrentPage}></Pager>
      </div>
    )
  }

  return (
    <main className="main main--search container-fluid">

      <div className="row">
        <div className="col-md-3">
          <div className="search-bar">
            <SearchBar postSearchHandler={postSearchHandler} q={q}>
</SearchBar>
          </div>
          <Facets facets={facets} filters={filters} setFilters={setFilters}>
</Facets>
        </div>
        {body}
      </div>
    </main>
  );
}
```

Azure Function: Suggestions from the catalog

The `Suggest` API takes a search term while a user is typing and suggests search terms such as book titles and authors across the documents in the search index, returning a small list of matches.

The search suggester, `sg`, is defined in the schema file used during bulk upload.

Routing for the Suggest API is contained in the function.json bindings.

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");
const { CONFIG } = require("../config");

// Create a SearchClient to send queries
const client = new SearchClient(
    `https://` + CONFIG.SearchServiceName + `.search.windows.net/`,
    CONFIG.SearchIndexName,
    new AzureKeyCredential(CONFIG.SearchApiKey)
);
```

```
module.exports = async function (context, req) {

    context.log(req);

    // Reading inputs from HTTP Request
    const q = (req.query.q || (req.body && req.body.q));
    const top = (req.query.top || (req.body && req.body.top));
    const suggester = (req.query.suggester || (req.body &&
req.body.suggester));

    // Let's get the top 5 suggestions for that search term
    const suggestions = await client.suggest(q, suggester, {top:
parseInt(top)});
    //const suggestions = await client.autocomplete(q, suggester, {top:
parseInt(top)});

    context.log(suggestions);

    context.res = {
        // status: 200, /* Defaults to 200 */
        headers: {
            "Content-type": "application/json"
        },
        body: { suggestions: suggestions.results}
    };
};
```

Client: Suggestions from the catalog

The Suggest function API is called in the React app at `\src\components\SearchBar\SearchBar.js` as part of component initialization:

JavaScript

```
import React, {useState, useEffect} from 'react';
import axios from 'axios';
import Suggestions from './Suggestions/Suggestions';

import "./SearchBar.css";

export default function SearchBar(props) {

    let [q, setQ] = useState("");
    let [suggestions, setSuggestions] = useState([]);
    let [showSuggestions, setShowSuggestions] = useState(false);

    const onSearchHandler = () => {
        props.postSearchHandler(q);
        setShowSuggestions(false);
    }
```

```
    const suggestionClickHandler = (s) => {
        document.getElementById("search-box").value = s;
        setShowSuggestions(false);
        props.postSearchHandler(s);
    }

    const onEnterButton = (event) => {
        if (event.keyCode === 13) {
            onSearchHandler();
        }
    }

    const onChangeHandler = () => {
        var searchTerm = document.getElementById("search-box").value;
        setShowSuggestions(true);
        setQ(searchTerm);

        // use this prop if you want to make the search more reactive
        if (props.searchChangeHandler) {
            props.searchChangeHandler(searchTerm);
        }
    }

    useEffect(_ =>{
        const timer = setTimeout(() => {
            const body = {
                q: q,
                top: 5,
                suggester: 'sg'
            };

            if (q === '') {
                setSuggestions([]);
            } else {
                axios.post( '/api/suggest', body)
                .then(response => {
                    console.log(JSON.stringify(response.data))
                    setSuggestions(response.data.suggestions);
                } )
                .catch(error => {
                    console.log(error);
                    setSuggestions([]);
                });
            }
        }, 300);
        return () => clearTimeout(timer);
    }, [q, props]);

    var suggestionDiv;
    if (showSuggestions) {
        suggestionDiv = (<Suggestions suggestions={suggestions}
suggestionClickHandler={(s) => suggestionClickHandler(s)}></Suggestions>);
    } else {
        suggestionDiv = (<div></div>);
    }
```

```
    return (
        <div >
            <div className="input-group" onKeyDown={e => onEnterButton(e)}>
                <div className="suggestions" >
                    <input
                        autoComplete="off" // setting for browsers; not the
app
                        type="text"
                        id="search-box"
                        className="form-control rounded-0"
                        placeholder="What are you looking for?"
                        onChange={onChangeHandler}
                        defaultValue={props.q}
                        onBlur={() => setShowSuggestions(false)}
                        onClick={() => setShowSuggestions(true)}>
                    </input>
                    {suggestionDiv}
                </div>
                <div className="input-group-btn">
                    <button className="btn btn-primary rounded-0"
type="submit" onClick={onSearchHandler}>
                        Search
                    </button>
                </div>
            </div>
        </div>
    );
};
```

Azure Function: Get specific document

The `Lookup` API takes an ID and returns the document object from the Search Index.

Routing for the Lookup API is contained in the function.json bindings.

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");
const { CONFIG } = require("../config");

// Create a SearchClient to send queries
const client = new SearchClient(
    `https://` + CONFIG.SearchServiceName + `.search.windows.net/`,
    CONFIG.SearchIndexName,
    new AzureKeyCredential(CONFIG.SearchApiKey)
);

module.exports = async function (context, req) {
```

```
    // Reading inputs from HTTP Request
    const id = (req.query.id || (req.body && req.body.id));

    // Returning the document with the matching id
    const document = await client.getDocument(id)

    context.log(document);

    context.res = {
        // status: 200, /* Defaults to 200 */
        headers: {
            "Content-type": "application/json"
        },
        body: { document: document}
    };

};
```

Client: Get specific document

This function API is called in the React app at `\src\pages\Details\Detail.js` as part of component initialization:

JavaScript

```
import React, { useState, useEffect } from "react";
import { useParams } from 'react-router-dom';
import Rating from '@material-ui/lab/Rating';
import CircularProgress from '@material-ui/core/CircularProgress';
import axios from 'axios';
import apiBaseUrl from "../../config";

import "./Details.css";

export default function Details() {

  let { id } = useParams();
  const [document, setDocument] = useState({});
  const [selectedTab, setTab] = useState(0);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setIsLoading(true);
    // console.log(id);
```

```
    axios.get(`${apiBaseUrl || ""}/api/lookup?id=${id}`)
      .then(response => {
        console.log(JSON.stringify(response.data))
        const doc = response.data.document;
        setDocument(doc);
        setIsLoading(false);
      })
      .catch(error => {
        console.log(error);
        setIsLoading(false);
      });
```

```
  }, [id]);

  // View default is loading with no active tab
  let detailsBody = (<CircularProgress />),
      resultStyle = "nav-link",
      rawStyle    = "nav-link";

  if (!isLoading && document) {
    // View result
    if (selectedTab === 0) {
      resultStyle += " active";
      detailsBody = (
        <div className="card-body">
          <h5 className="card-title">{document.original_title}</h5>
          <img className="image" src={document.image_url} alt="Book cover">
</img>
          <p className="card-text">{document.authors?.join('; ')} -
{document.original_publication_year}</p>
          <p className="card-text">ISBN {document.isbn}</p>
          <Rating name="half-rating-read" value=
{parseInt(document.average_rating)} precision={0.1} readOnly></Rating>
          <p className="card-text">{document.ratings_count} Ratings</p>
        </div>
      );
    }

    // View raw data
    else {
      rawStyle += " active";
      detailsBody = (
        <div className="card-body text-left">
          <pre><code>
            {JSON.stringify(document, null, 2)}
          </code></pre>
        </div>
      );
    }
  }

  return (
    <main className="main main--details container fluid">
      <div className="card text-center result-container">
        <div className="card-header">
```

```
          <ul className="nav nav-tabs card-header-tabs">
              <li className="nav-item"><button className={resultStyle}
onClick={() => setTab(0)}>Result</button></li>
              <li className="nav-item"><button className={rawStyle} onClick=
{() => setTab(1)}>Raw Data</button></li>
          </ul>
        </div>
        {detailsBody}
      </div>
    </main>
  );
}
```

Next steps

- Index Azure SQL data

1 - Overview of adding search to a website with Python

Article • 11/23/2022 • 2 minutes to read

This tutorial builds a website to search through a catalog of books then deploys the website to an Azure Static Web App.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

What does the sample do?

This sample website provides access to a catalog of 10,000 books. A user can search the catalog by entering text in the search bar. While the user enters text, the website uses the search index's suggest feature to complete the text. Once the query finishes, the list of books is displayed with a portion of the details. A user can select a book to see all the details, stored in the search index, of the book.

The search experience includes:

- Search – provides search functionality for the application.
- Suggest – provides suggestions as the user is typing in the search bar.
- Document Lookup – looks up a document by ID to retrieve all of its contents for the details page.

How is the sample organized?

The sample includes the following:

App	Purpose	GitHub Repository Location
Client	React app (presentation layer) to display books, with search. It calls the Azure Function app.	/search-website-functions-v4/client
Server	Azure Function app (business layer) - calls the Azure Cognitive Search API using Python SDK	/search-website-functions-v4/api
Bulk insert	Python file to create the index and add documents to it.	/search-website-functions-v4/bulk-upload

Set up your development environment

Install the following for your local development environment.

- Python 3.9
- Git
- Visual Studio Code and the following extensions
  - Azure Static Web App
- Optional:
  - This tutorial doesn't run the Azure Function API locally but if you intend to run it locally, you need to install azure-functions-core-tools.

Fork and clone the search sample with git

Forking the sample repository is critical to be able to deploy the static web app. The web apps determine the build actions and deployment content based on your own GitHub fork location. Code execution in the Static Web App is remote, with Azure static web apps reading from the code in your forked sample.

1. On GitHub, fork the sample repository.

   Complete the fork process in your web browser with your GitHub account. This tutorial uses your fork as part of the deployment to an Azure Static Web App.

2. At a bash terminal, download the sample application to your local computer.

   Replace `YOUR-GITHUB-ALIAS` with your GitHub alias.

```bash
git clone https://github.com/YOUR-GITHUB-ALIAS/azure-search-python-samples
```

3. In Visual Studio Code, open your local folder of the cloned repository. The remaining tasks are accomplished from Visual Studio Code, unless specified.

Create a resource group for your Azure resources

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the **Resources** section, select Add (+), and then select **Create Resource Group**.

3. Enter a resource group name, such as `cognitive-search-website-tutorial`.

4. Select a location close to you.

5. When you create the Cognitive Search and Static Web Apps resources, later in the tutorial, use this resource group.

   Creating a resource group gives you a logical unit to manage the resources, including deleting them when you're finished using them.

Next steps

- Create a Search Index and load with documents

- Deploy your Static Web App

2 - Create and load Search Index with Python

Article • 01/23/2023 • 5 minutes to read

Continue to build your search-enabled website by following these steps:

- Create a search resource
- Create a new index
- Import data with Python using the sample script and Azure SDK azure-search-documents.

Create an Azure Cognitive Search resource

Create a new search resource using PowerShell and the **Az.Search** module. In this section, you'll also create a query key used for read-access to the index, and get the built-in admin key used for adding objects.

1. In Visual Studio Code, open a new terminal window.

2. Connect to Azure:

   PowerShell

   ```
   Connect-AzAccount -TenantID <your-tenant-ID>
   ```

   > **Note**
   >
   > You might need to provide a tenant ID, which you can find in the Azure portal in **Portal settings** > **Directories + subscriptions**.

3. Before creating a new search service, you can list existing search services for your subscription to see if there's one you want to use:

   PowerShell

   ```
   Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
   ```

4. Load the **Az.Search** module:

   PowerShell

```
Install-Module -Name Az.Search
```

5. Create a new search service. Use the following cmdlet as a template, substituting valid values for the resource group, service name, tier, region, partitions, and replicas:

PowerShell

```
New-AzSearchService -ResourceGroupName "my resource group"  -Name
"myDemoSearchSvc" -Sku "Free" -Location "West US" -PartitionCount 1 -
ReplicaCount 1 -HostingMode Default
```

Prompt	Enter
Enter a globally unique name for the new search service.	**Remember this name**. This resource name becomes part of your resource endpoint.
Select a resource group for new resources	Use the resource group you created for this tutorial.
Select the SKU for your search service.	Use **Free** for this tutorial. You can't change a SKU pricing tier after the service is created.
Select a location for new resources.	Select a region close to you.

6. Create a query key that grants read access to a search service. Query keys have to be explicitly created. Copy the query key to Notepad so that you can paste it into the client code in a later step:

PowerShell

```
New-AzSearchQueryKey -ResourceGroupName "my resource group"  -
ServiceName "myDemoSearchSvc" -Name "mySrchQueryKey"
```

7. Get the search service admin API key that was automatically created for your search service. An admin API key provides write access to the search service. Copy either one of the admin keys to Notepad so that you can use it in the bulk import step that creates and loads an index:

PowerShell

```
Get-AzSearchAdminKeyPair  -ResourceGroupName "my resource group" -
ServiceName "myDemoSearchSvc"
```

Prepare the bulk import script for Search

The script uses the Azure SDK for Cognitive Search:

- PYPI package azure-search-documents
- Reference Documentation

1. In Visual Studio Code, open the `bulk_upload.py` file in the subdirectory, `search-website-functions-v4/bulk-upload`, replace the following variables with your own values to authenticate with the Azure Search SDK:

   - YOUR-SEARCH-SERVICE-NAME
   - YOUR-SEARCH-SERVICE-ADMIN-API-KEY

   Python

   ```python
   import sys
   import json
   import requests
   import pandas as pd
   from azure.core.credentials import AzureKeyCredential
   from azure.search.documents import SearchClient
   from azure.search.documents.indexes import SearchIndexClient
   from azure.search.documents.indexes.models import SearchIndex
   from azure.search.documents.indexes.models import (
       ComplexField,
       CorsOptions,
       SearchIndex,
       ScoringProfile,
       SearchFieldDataType,
       SimpleField,
       SearchableField,
   )

   # Get the service name (short name) and admin API key from the
   environment
   service_name = "YOUR-SEARCH-SERVICE-NAME"
   key = "YOUR-SEARCH-SERVICE-ADMIN-API-KEY"
   endpoint = "https://{}.search.windows.net/".format(service_name)

   # Give your index a name
   # You can also supply this at runtime in __main__
   index_name = "good-books"

   # Search Index Schema definition
   index_schema = "./good-books-index.json"

   # Books catalog
   books_url = "https://raw.githubusercontent.com/zygmuntz/goodbooks-
   10k/master/books.csv"
   batch_size = 1000
   ```

```python
# Instantiate a client
class CreateClient(object):
    def __init__(self, endpoint, key, index_name):
        self.endpoint = endpoint
        self.index_name = index_name
        self.key = key
        self.credentials = AzureKeyCredential(key)

    # Create a SearchClient
    # Use this to upload docs to the Index
    def create_search_client(self):
        return SearchClient(
            endpoint=self.endpoint,
            index_name=self.index_name,
            credential=self.credentials,
        )

    # Create a SearchIndexClient
    # This is used to create, manage, and delete an index
    def create_admin_client(self):
        return SearchIndexClient(endpoint=endpoint,
credential=self.credentials)


# Get Schema from File or URL
def get_schema_data(schema, url=False):
    if not url:
        with open(schema) as json_file:
            schema_data = json.load(json_file)
            return schema_data
    else:
        data_from_url = requests.get(schema)
        schema_data = json.loads(data_from_url.content)
        return schema_data


# Create Search Index from the schema
# If reading the schema from a URL, set url=True
def create_schema_from_json_and_upload(schema, index_name,
admin_client, url=False):

    cors_options = CorsOptions(allowed_origins=["*"],
max_age_in_seconds=60)
    scoring_profiles = []
    schema_data = get_schema_data(schema, url)

    index = SearchIndex(
        name=index_name,
        fields=schema_data["fields"],
        scoring_profiles=scoring_profiles,
        suggesters=schema_data["suggesters"],
        cors_options=cors_options,
```

```python
# Create Search Index from the schema
# If reading the schema from a URL, set url=True
def create_schema_from_json_and_upload(schema, index_name,
admin_client, url=False):

    cors_options = CorsOptions(allowed_origins=["*"],
max_age_in_seconds=60)
    scoring_profiles = []
    schema_data = get_schema_data(schema, url)

    index = SearchIndex(
        name=index_name,
        fields=schema_data["fields"],
        scoring_profiles=scoring_profiles,
        suggesters=schema_data["suggesters"],
        cors_options=cors_options,
    )

    try:
        upload_schema = admin_client.create_index(index)
        if upload_schema:
            print(f"Schema uploaded; Index created for {index_name}.")
        else:
            exit(0)
    except:
        print("Unexpected error:", sys.exc_info()[0])


# Convert CSV data to JSON
def convert_csv_to_json(url):
    df = pd.read_csv(url)
    convert = df.to_json(orient="records")
    return json.loads(convert)


# Batch your uploads to Azure Search
def batch_upload_json_data_to_index(json_file, client):
    batch_array = []
    count = 0
    batch_counter = 0
    for i in json_file:
        count += 1
        batch_array.append(
            {
                "id": str(i["book_id"]),
                "goodreads_book_id": int(i["goodreads_book_id"]),
                "best_book_id": int(i["best_book_id"]),
                "work_id": int(i["work_id"]),
                "books_count": i["books_count"] if i["books_count"]
else 0,
                "isbn": str(i["isbn"]),
                "isbn13": str(i["isbn13"]),
                "authors": i["authors"].split(",") if i["authors"] else
None,
                "original_publication_year":
```

```
int(i["original_publication_year"])
                if i["original_publication_year"]
                else 0,
                "original_title": i["original_title"],
                "title": i["title"],
                "language_code": i["language_code"],
                "average_rating": int(i["average_rating"])
                if i["average_rating"]
                else 0,
                "ratings_count": int(i["ratings_count"]) if
i["ratings_count"] else 0,
                "work_ratings_count": int(i["work_ratings_count"])
                if i["work_ratings_count"]
                else 0,
                "work_text_reviews_count": i["work_text_reviews_count"]
                if i["work_text_reviews_count"]
                else 0,
                "ratings_1": int(i["ratings_1"]) if i["ratings_1"] else
0,
                "ratings_2": int(i["ratings_2"]) if i["ratings_2"] else
0,
                "ratings_3": int(i["ratings_3"]) if i["ratings_3"] else
0,
                "ratings_4": int(i["ratings_4"]) if i["ratings_4"] else
0,
                "ratings_5": int(i["ratings_5"]) if i["ratings_5"] else
0,
                "image_url": i["image_url"],
                "small_image_url": i["small_image_url"],
            }
        )

        # In this sample, we limit batches to 1000 records.
        # When the counter hits a number divisible by 1000, the batch
is sent.
        if count % batch_size == 0:
            client.upload_documents(documents=batch_array)
            batch_counter += 1
            print(f"Batch sent! - #{batch_counter}")
            batch_array = []

    # This will catch any records left over, when not divisible by 1000
    if len(batch_array) > 0:
        client.upload_documents(documents=batch_array)
        batch_counter += 1
        print(f"Final batch sent! - #{batch_counter}")

    print("Done!")


if __name__ == "__main__":
    start_client = CreateClient(endpoint, key, index_name)
    admin_client = start_client.create_admin_client()
    search_client = start_client.create_search_client()
    schema = create_schema_from_json_and_upload(
```

```
        index_schema, index_name, admin_client, url=False
    )
    books_data = convert_csv_to_json(books_url)
    batch_upload = batch_upload_json_data_to_index(books_data,
search_client)
    print("Upload complete")
```

2. Open an integrated terminal in Visual Studio for the project directory's subdirectory, `search-website-functions-v4/bulk-upload`, and run the following command to install the dependencies.

macOS/Linux

Bash

```
python3 -m pip install -r requirements.txt
```

Run the bulk import script for Search

1. Continue using the integrated terminal in Visual Studio for the project directory's subdirectory, `search-website-functions-v4/bulk-upload`, to run the following bash command to run the `bulk_upload.py` script:

macOS/Linux

Bash

```
python3 bulk-upload.py
```

2. As the code runs, the console displays progress.

3. When the upload is complete, the last statement printed to the console is "Done! Upload complete".

Review the new Search Index

Once the upload completes, the search index is ready to use. Review your new index in Azure portal.

1. In Azure portal, find the search service you created in the previous step.

2. On the Overview page, select the **Indexes** tab in the middle of the page, and then select the index you created in the previous step.

3. By default, the index opens in the **Search explorer** tab. Select **Search** to return documents from the index.

Rollback bulk import file changes

Use the following git command in the Visual Studio Code integrated terminal at the `bulk-insert` directory, to roll back the changes. They aren't needed to continue the tutorial and you shouldn't save or push these secrets to your repo.

```git
git checkout .
```

Copy your Search resource name

Note your **Search resource name**. You'll need this to connect the Azure Function app to your search resource.

> **⊗ Caution**
>
> While you may be tempted to use your search admin key in the Azure Function, that isn't following the principle of least privilege. The Azure Function will use the query key to conform to least privilege.

Next steps

Deploy your Static Web App

3 - Deploy the search-enabled Python website

Article • 11/02/2022 • 6 minutes to read

Deploy the search-enabled website as an Azure Static Web Apps site. This deployment includes both the React app and the Function app.

The Static Web app pulls the information and files for deployment from GitHub using your fork of the samples repository.

Create a Static Web App in Visual Studio Code

1. Select **Azure** from the Activity Bar, then open **Resources** from the Side bar.

2. Right-click **Static Web Apps** and then select **Create Static Web App (Advanced)**.

3. If you see a pop-up window in VS Code asking which branch you want to deploy from, select the default branch, usually **master** or **main**.

This setting means only changes you commit to that branch are deployed to your static web app.

4. If you see a pop-up window asking you to commit your changes, don't do this. The secrets from the bulk import step shouldn't be committed to the repository.

   To roll back the changes, in VS Code select the Source Control icon in the Activity bar, then select each changed file in the Changes list and select the **Discard changes** icon.

5. Follow the prompts to provide the following information:

Prompt	Enter
Select a resource group for new resources.	Use the resource group you created for this tutorial.
Enter the name for the new Static Web App.	Create a unique name for your resource. For example, you can prepend your name to the repository name such as, `joansmith-azure-search-dotnet-samples`.
Select a SKU	Select the free SKU for this tutorial.
Select a location for new resources.	Select a region close to you.
Choose build preset to configure default project structure.	Select **Custom**
Select the location of your application code	`search-website-functions-v4/client` This is the path, from the root of the repository, to your static web app.
Enter the path of your build output...	`build` This is the path, from your static web app, to your generated files.

6. The resource is created and a notification window appears.

   When the resource is created, it creates a GitHub action file on GitHub.

7. Select **Open Actions in GitHub** from the Notifications. This opens a browser window pointed to your forked repo.

   Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

8. Pull the new GitHub action file to your local computer by synchronizing your local fork with your remote fork:

```bash
git pull origin main
```

   - *origin* refers to your forked repo.
   - *main* refers to the default branch.

9. In Visual Studio file explorer, find and open the workflow file in the `./.github/workflows/` directory. The file path and name looks *something* `.github\workflows\azure-static-web-apps-lemon-mushroom-0e1bd060f.yml`.

   The *part* of the YAML file relevant to the static web app is shown below:

```yml
name: Azure Static Web Apps CI/CD

on:
  push:
    branches:
      - main
  pull_request:
    types: [opened, synchronize, reopened, closed]
    branches:
      - main

jobs:
  build_and_deploy_job:
    if: github.event_name == 'push' || (github.event_name ==
'pull_request' && github.event.action != 'closed')
    runs-on: ubuntu-latest
    name: Build and Deploy Job
    steps:
      - uses: actions/checkout@v2
        with:
          submodules: true
      - name: Build And Deploy
        id: builddeploy
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          repo_token: ${{ secrets.GITHUB_TOKEN }} # Used for Github
integrations (i.e. PR comments)
          action: "upload"
```

```
          ###### Repository/Build Configurations - These values can be
configured to match your app requirements. ######
          # For more information regarding Static Web App workflow
configurations, please visit: https://aka.ms/swaworkflowconfig
          app_location: "search-website-functions-v4/client" # App
source code path
          api_location: "search-website-functions-v4/api" # Api source
code path - optional
          output_location: "build" # Built app content directory -
optional
          ###### End of Repository/Build Configurations ######
```

```yml
  close_pull_request_job:
    if: github.event_name == 'pull_request' && github.event.action ==
'closed'
    runs-on: ubuntu-latest
    name: Close Pull Request Job
    steps:
      - name: Close Pull Request
        id: closepullrequest
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{
secrets.AZURE_STATIC_WEB_APPS_API_TOKEN_RANDOM_WORDS_1234 }}
          action: "close"
```

10. Edit your action file to contain the `api_location` property. If your local file doesn't have the property, add it below the `app_location` property.

```yml
api_location: "search-website-functions-v4/api"
```

11. Commit changes to your local repository.

```Bash
git add *.yml && git commit -m "update action for static web app"
```

12. Push changes to GitHub.

```Bash
git push origin main
```

The updated action in your remote fork creates a new build and deploy to your static web app. Wait until the *workflow* completes before continuing. This may take a minute or two to finish.

Get Cognitive Search query key in Visual Studio Code

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.

2. In the Side bar, select your Azure subscription under the **Azure: Cognitive Search** area, then right-click on your Search resource and select **Copy Query Key**.

3. Keep this query key, you'll need to use it in the next section. The query key is able to query your Index.

Add configuration settings in Azure portal

The Azure Function app won't return Search data until the Search secrets are in settings.

1. Select **Azure** from the Activity Bar.

2. Right-click on your Static Web Apps resource then select **Open in Portal**.

3. Select **Configuration** then select **+ Add**.

4. Add each of the following settings:

Setting	Your Search resource value
SearchApiKey	Your Search query key
SearchServiceName	Your Search resource name
SearchIndexName	`good-books`
SearchFacets	`authors*,language_code`

Azure Cognitive Search requires different syntax for filtering collections than it does for strings. Add a `*` after a field name to denote that the field is of type

`Collection(Edm.String)`. This allows the Azure Function to add filters correctly to queries.

5. Select **Save** to save the settings.

6. Return to VS Code.

7. Refresh your static web app to see the application settings.

Use search in your static web app

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Static Web Apps` area and find the static web app you created for this tutorial.
3. Right-click the static web app name and select **Browse site**.

4. Select **Open** in the pop-up dialog.

5. In the website search bar, enter a search query such as `code`, so the suggest feature suggests book titles. Select a suggestion or continue entering your own query. Press enter when you've completed your search query.

6. Review the results then select one of the books to see more details.

Troubleshooting

If the web app didn't deploy or work, use the following list to determine and fix the issue:

- **Did the deployment succeed?**

  In order to determine if your deployment succeeded, you need to go to *your* fork of the sample repo and review the success or failure of the GitHub action. There should be only 1 action and it should have static web app settings for the

`app_location`, `api_location`, and `output_location`. If the action didn't deploy successfully, dive into the action logs and look for the last failure.

- **Does the client (front-end) application work?**

  You should be able to get to your web app and it should successfully display. If the deployment succeeded but the website doesn't display, this may be an issue with how the static web app is configured for rebuilding the app, once it is on Azure.

- **Does the API (serverless back-end) application work?**

  You should be able to interact with the client app, searching for books and filtering. If the form doesn't return any values, open the browser's developer tools, and determine if the HTTP calls to the API were successful. If the calls weren't successful, the most likely reason if the static web app configurations for the API endpoint name and Search query key are incorrect.

Clean up resources

To clean up the resources created in this tutorial, delete the resource group.

1. In Visual Studio Code, open the Activity bar, and select the Azure icon.
2. In the Side bar, **right-click on your Azure subscription** under the `Resource Groups` area and find the resource group you created for this tutorial.
3. Right-click the resource group name then select **Delete**. This deletes both the Search and Static Web Apps resources.
4. If you no longer want the GitHub fork of the sample, remember to delete that on GitHub. Go to your fork's **Settings** then delete the fork.

Next steps

- Understand Search integration for the search-enabled website

---

Additional resources

🕮 Documentation

Python tutorial: Search integration overview - Azure Cognitive Search

Technical overview and setup for adding search to a website with Python and deploying to Azure

Static Web App.

Azure Cognitive Search client library for Python

Quickstart: Create a search index in Python - Azure Cognitive Search

Learn how to create a search index, load data, and run queries using Python, Jupyter Notebook, and the Azure.Documents.Search client library for Python.

Python tutorial: Add search to web apps - Azure Cognitive Search

Create index and import CSV data into Search index with Python using the PYPI package SDK azure-search-documents.

azure.search.documents.SearchClient class

A client to interact with an existing Azure search index.

Content metadata properties - Azure Cognitive Search

Metadata properties can provide content to fields in a search index. This article lists metadata properties supported in Azure Cognitive Search.

Python tutorial: AI on Azure blobs - Azure Cognitive Search

Step through an example of text extraction and natural language processing over content in Blob storage using a Jupyter Python notebook and the Azure Cognitive Search REST APIs.

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Show 5 more

Training

Learning path

Azure Static Web Apps - Training

Azure Static Web Apps is a modern web app service that offers streamlined full-stack development from source code to global high availability.

4 - Python Search integration cheat sheet

Article • 11/02/2022 • 8 minutes to read

In the previous lessons, you added search to a Static Web App. This lesson highlights the essential steps that establish integration. If you're looking for a cheat sheet on how to integrate search into your Python app, this article explains what you need to know.

The application is available:

- Sample
- Demo website - aka.ms/azs-good-books

Azure SDK azure-search-documents

The Function app uses the Azure SDK for Cognitive Search:

- PYPI package azure-search-documents
- Reference Documentation

The Function app authenticates through the SDK to the cloud-based Cognitive Search API using your resource name, API key, and index name. The secrets are stored in the Static Web App settings and pulled in to the Function as environment variables.

Configure secrets in a configuration file

The Azure Function app settings environment variables are pulled in from a file, `__init__.py`, shared between the three API functions.

Python

```python
import os


def azure_config():

    configs = {}
    configs["search_facets"] = os.environ.get("SearchFacets", "")
    configs["search_index_name"] = os.environ.get("SearchIndexName", "")
    configs["search_service_name"] = os.environ.get("SearchServiceName", "")
    configs["search_api_key"] = os.environ.get("SearchApiKey", "")

    return configs
```

Azure Function: Search the catalog

The Search API takes a search term and searches across the documents in the Search Index, returning a list of matches.

Routing for the Search API is contained in the function.json bindings.

The Azure Function pulls in the search configuration information, and fulfills the query.

Python

```python
import logging
import azure.functions as func
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from shared_code import azure_config
import json

environment_vars = azure_config()

# Set Azure Search endpoint and key
endpoint =
f'https://{environment_vars["search_service_name"]}.search.windows.net'
key = environment_vars["search_api_key"]

# Your index name
index_name = "good-books"

# Create Azure SDK client
search_client = SearchClient(endpoint, index_name, AzureKeyCredential(key))

# returns obj like {authors: 'array', language_code:'string'}
def read_facets(facetsString):
    facets = facetsString.split(",")
    output = {}
    for x in facets:
        if x.find("*") != -1:
            newVal = x.replace("*", "")
            output[newVal] = "array"
        else:
            output[x] = "string"

    return output


# creates filters in odata syntax
def create_filter_expression(filter_list, facets):
    i = 0
    filter_expressions = []
    return_string = ""
    separator = " and "
```

```
    while i < len(filter_list):
        field = filter_list[i]["field"]
        value = filter_list[i]["value"]

        if facets[field] == "array":
            print("array")
            filter_expressions.append(f"{field}/any(t: search.in(t,
'{value}', ','))")
        else:
            print("value")
            filter_expressions.append(f"{field} eq '{value}'")

        i += 1

    return_string = separator.join(filter_expressions)

    return return_string


def new_shape(docs):

    old_api_shape = list(docs)

    count = 0
    client_side_expected_shape = []

    for item in old_api_shape:

        new_document = {}
        new_document["score"] = item["@search.score"]
        new_document["highlights"] = item["@search.highlights"]

        new_shape = {}
        new_shape["id"] = item["id"]
        new_shape["goodreads_book_id"] = item["goodreads_book_id"]
        new_shape["best_book_id"] = item["best_book_id"]
        new_shape["work_id"] = item["work_id"]
        new_shape["books_count"] = item["books_count"]
        new_shape["isbn"] = item["isbn"]
        new_shape["isbn13"] = item["isbn13"]
        new_shape["authors"] = item["authors"]
        new_shape["original_publication_year"] =
item["original_publication_year"]
        new_shape["original_title"] = item["original_title"]
        new_shape["title"] = item["title"]
        new_shape["language_code"] = item["language_code"]
        new_shape["average_rating"] = item["average_rating"]
        new_shape["ratings_count"] = item["ratings_count"]
        new_shape["work_ratings_count"] = item["work_ratings_count"]
        new_shape["work_text_reviews_count"] =
item["work_text_reviews_count"]
        new_shape["ratings_1"] = item["ratings_1"]
        new_shape["ratings_2"] = item["ratings_2"]
        new_shape["ratings_3"] = item["ratings_3"]
```

```
        new_shape["ratings_4"] = item["ratings_4"]
        new_shape["ratings_5"] = item["ratings_5"]
        new_shape["image_url"] = item["image_url"]
        new_shape["small_image_url"] = item["small_image_url"]

        new_document["document"] = new_shape

        client_side_expected_shape.append(new_document)

    return list(client_side_expected_shape)


def main(req: func.HttpRequest) -> func.HttpResponse:

    # variables sent in body
    req_body = req.get_json()
    q = req_body.get("q")
    top = req_body.get("top") or 8
    skip = req_body.get("skip") or 0
    filters = req_body.get("filters") or []

    facets = environment_vars["search_facets"]
    facetKeys = read_facets(facets)

    filter = ""
    if len(filters):
        filter = create_filter_expression(filters, facetKeys)

    if q:
        logging.info(f"/Search q = {q}")

        search_results = search_client.search(
            search_text=q,
            top=top,
            skip=skip,
            facets=facetKeys,
            filter=filter,
            include_total_count=True,
        )

        returned_docs = new_shape(search_results)
        returned_count = search_results.get_count()
        returned_facets = search_results.get_facets()

        # format the React app expects
        full_response = {}

        full_response["count"] = search_results.get_count()
        full_response["facets"] = search_results.get_facets()
        full_response["results"] = returned_docs

        return func.HttpResponse(
            body=json.dumps(full_response), mimetype="application/json",
status_code=200
```

```
        )
    else:
        return func.HttpResponse("No query param found.", status_code=200)
```

Client: Search from the catalog

Call the Azure Function in the React client with the following code.

JavaScript

```
import React, { useEffect, useState } from 'react';
import axios from 'axios';
import CircularProgress  from '@material-ui/core/CircularProgress';
import { useLocation, useNavigate } from "react-router-dom";

import Results from '../../components/Results/Results';
import Pager from '../../components/Pager/Pager';
import Facets from '../../components/Facets/Facets';
import SearchBar from '../../components/SearchBar/SearchBar';

import "./Search.css";

export default function Search() {

  let location = useLocation();
  const navigate = useNavigate();

  const [ results, setResults ] = useState([]);
  const [ resultCount, setResultCount ] = useState(0);
  const [ currentPage, setCurrentPage ] = useState(1);
  const [ q, setQ ] = useState(new URLSearchParams(location.search).get('q')
?? "*");
  const [ top ] = useState(new URLSearchParams(location.search).get('top')
?? 8);
  const [ skip, setSkip ] = useState(new
URLSearchParams(location.search).get('skip') ?? 0);
  const [ filters, setFilters ] = useState([]);
  const [ facets, setFacets ] = useState({});
  const [ isLoading, setIsLoading ] = useState(true);

  let resultsPerPage = top;

  useEffect(() => {
    setIsLoading(true);
    setSkip((currentPage-1) * top);
    const body = {
      q: q,
      top: top,
      skip: skip,
      filters: filters
    };
```

```
    axios.post( '/api/search', body)
      .then(response => {
            console.log(JSON.stringify(response.data))
            setResults(response.data.results);
            setFacets(response.data.facets);
            setResultCount(response.data.count);
            setIsLoading(false);
        } )
        .catch(error => {
            console.log(error);
            setIsLoading(false);
        });

  }, [q, top, skip, filters, currentPage]);
```

```
  // pushing the new search term to history when q is updated
  // allows the back button to work as expected when coming back from the
details page
  useEffect(() => {
    navigate('/search?q=' + q);
    setCurrentPage(1);
    setFilters([]);
    // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [q]);


  let postSearchHandler = (searchTerm) => {
    //console.log(searchTerm);
    setQ(searchTerm);
  }

  var body;
  if (isLoading) {
    body = (
      <div className="col-md-9">
        <CircularProgress />
      </div>);
  } else {
    body = (
      <div className="col-md-9">
        <Results documents={results} top={top} skip={skip} count=
{resultCount}></Results>
        <Pager className="pager-style" currentPage={currentPage}
resultCount={resultCount} resultsPerPage={resultsPerPage} setCurrentPage=
{setCurrentPage}></Pager>
      </div>
    )
  }

  return (
    <main className="main main--search container-fluid">

      <div className="row">
        <div className="col-md-3">
          <div className="search-bar">
```

```
            <SearchBar postSearchHandler={postSearchHandler} q={q}>
</SearchBar>
          </div>
          <Facets facets={facets} filters={filters} setFilters={setFilters}>
</Facets>
        </div>
        {body}
      </div>
    </main>
  );
}
```

Azure Function: Suggestions from the catalog

The `Suggest` API takes a search term while a user is typing and suggests search terms such as book titles and authors across the documents in the search index, returning a small list of matches.

The search suggester, `sg`, is defined in the schema file used during bulk upload.

Routing for the Suggest API is contained in the function.json bindings.

Python

```python
import logging
import azure.functions as func
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from shared_code import azure_config
import json

environment_vars = azure_config()

# curl --header "Content-Type: application/json" \
#  --request POST \
#  --data '{"q":"code","top":"5", "suggester":"sg"}' \
#  http://localhost:7071/api/Suggest

# Set Azure Search endpoint and key
service_name = environment_vars["search_service_name"]
endpoint = f"https://{service_name}.search.windows.net"
key = environment_vars["search_api_key"]

# Your index name
index_name = "good-books"

# Create Azure SDK client
search_client = SearchClient(endpoint, index_name, AzureKeyCredential(key))
```

```
def main(req: func.HttpRequest) -> func.HttpResponse:

    # variables sent in body
    req_body = req.get_json()
    q = req_body.get("q")
    top = req_body.get("top")
    suggester = req_body.get("suggester")

    if q:
        logging.info(f"/Suggest q = {q}")
        suggestions = search_client.suggest(search_text=q,
suggester_name="sg", top=5)

        # format the React app expects
        full_response = {}
        full_response["suggestions"] = suggestions
        print(suggestions)

        return func.HttpResponse(
            body=json.dumps(full_response), mimetype="application/json",
status_code=200
        )
    else:
        return func.HttpResponse("No query param found.", status_code=200)
```

Client: Suggestions from the catalog

The Suggest function API is called in the React app at `client\src\components\SearchBar\SearchBar.js` as part of component initialization:

JavaScript

```
import React, {useState, useEffect} from 'react';
import axios from 'axios';
import Suggestions from './Suggestions/Suggestions';

import "./SearchBar.css";

export default function SearchBar(props) {

    let [q, setQ] = useState("");
    let [suggestions, setSuggestions] = useState([]);
    let [showSuggestions, setShowSuggestions] = useState(false);

    const onSearchHandler = () => {
        props.postSearchHandler(q);
        setShowSuggestions(false);
    }

    const suggestionClickHandler = (s) => {
        document.getElementById("search-box").value = s;
```

```
        setShowSuggestions(false);
        props.postSearchHandler(s);
    }

    const onEnterButton = (event) => {
        if (event.keyCode === 13) {
            onSearchHandler();
        }
    }

    const onChangeHandler = () => {
        var searchTerm = document.getElementById("search-box").value;
        setShowSuggestions(true);
        setQ(searchTerm);

        // use this prop if you want to make the search more reactive
        if (props.searchChangeHandler) {
            props.searchChangeHandler(searchTerm);
        }
    }

    useEffect(_ =>{
        const timer = setTimeout(() => {
            const body = {
                q: q,
                top: 5,
                suggester: 'sg'
            };

            if (q === '') {
                setSuggestions([]);
            } else {
                axios.post( '/api/suggest', body)
                .then(response => {
                    console.log(JSON.stringify(response.data))
                    setSuggestions(response.data.suggestions);
                } )
                .catch(error => {
                    console.log(error);
                    setSuggestions([]);
                });
            }
        }, 300);
        return () => clearTimeout(timer);
    }, [q, props]);

    var suggestionDiv;
    if (showSuggestions) {
        suggestionDiv = (<Suggestions suggestions={suggestions}
suggestionClickHandler={(s) => suggestionClickHandler(s)}></Suggestions>);
    } else {
        suggestionDiv = (<div></div>);
    }

    return (
```

```
        <div >
            <div className="input-group" onKeyDown={e => onEnterButton(e)}>
                <div className="suggestions" >
                    <input
                        autoComplete="off" // setting for browsers; not the
app
                        type="text"
                        id="search-box"
                        className="form-control rounded-0"
                        placeholder="What are you looking for?"
                        onChange={onChangeHandler}
                        defaultValue={props.q}
                        onBlur={() => setShowSuggestions(false)}
                        onClick={() => setShowSuggestions(true)}>
                    </input>
                    {suggestionDiv}
                </div>
                <div className="input-group-btn">
                    <button className="btn btn-primary rounded-0"
type="submit" onClick={onSearchHandler}>
                        Search
                    </button>
                </div>
            </div>
        </div>
    );
};
```

Azure Function: Get specific document

The `Lookup` API takes an ID and returns the document object from the Search Index.

Routing for the Lookup API is contained in the function.json bindings.

Python

```
import logging
import azure.functions as func
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
from shared_code import azure_config
import json
```

```
environment_vars = azure_config()

# Set Azure Search endpoint and key
endpoint =
f'https://{environment_vars["search_service_name"]}.search.windows.net'
key = environment_vars["search_api_key"]

# Your index name
index_name = "good-books"

# Create Azure SDK client
search_client = SearchClient(endpoint, index_name, AzureKeyCredential(key))


def main(req: func.HttpRequest) -> func.HttpResponse:

    # http://localhost:7071/api/Lookup?id=100
    docid = req.params.get("id")

    if docid:
        logging.info(f"/Lookup id = {docid}")
        returnedDocument = search_client.get_document(key=docid)

        full_response = {}
        full_response["document"] = returnedDocument

        return func.HttpResponse(
            body=json.dumps(full_response), mimetype="application/json",
status_code=200
        )
    else:
        return func.HttpResponse("No doc id param found.", status_code=200)
```

Client: Get specific document

This function API is called in the React app at `client\src\pages\Details\Detail.js` as part of component initialization:

JavaScript

```
import React, { useState, useEffect } from "react";
import { useParams } from 'react-router-dom';
import Rating from '@material-ui/lab/Rating';
import CircularProgress from '@material-ui/core/CircularProgress';
import axios from 'axios';

import "./Details.css";

export default function Details() {

  let { id } = useParams();
```

```
  const [document, setDocument] = useState({});
  const [selectedTab, setTab] = useState(0);
  const [isLoading, setIsLoading] = useState(true);

  useEffect(() => {
    setIsLoading(true);
    // console.log(id);
    axios.get('/api/lookup?id=' + id)
      .then(response => {
        console.log(JSON.stringify(response.data))
        const doc = response.data.document;
        setDocument(doc);
        setIsLoading(false);
      })
      .catch(error => {
        console.log(error);
        setIsLoading(false);
      });

  }, [id]);

  // View default is loading with no active tab
  let detailsBody = (<CircularProgress />),
      resultStyle = "nav-link",
      rawStyle    = "nav-link";

  if (!isLoading && document) {
    // View result
    if (selectedTab === 0) {
      resultStyle += " active";
      detailsBody = (
        <div className="card-body">
          <h5 className="card-title">{document.original_title}</h5>
          <img className="image" src={document.image_url} alt="Book cover">
</img>
          <p className="card-text">{document.authors?.join('; ')} -
{document.original_publication_year}</p>
          <p className="card-text">ISBN {document.isbn}</p>
          <Rating name="half-rating-read" value=
{parseInt(document.average_rating)} precision={0.1} readOnly></Rating>
          <p className="card-text">{document.ratings_count} Ratings</p>
        </div>
      );
    }

    // View raw data
    else {
      rawStyle += " active";
      detailsBody = (
        <div className="card-body text-left">
          <pre><code>
            {JSON.stringify(document, null, 2)}
          </code></pre>
        </div>
      );
```

```
    }
  }

  return (
    <main className="main main--details container fluid">
      <div className="card text-center result-container">
        <div className="card-header">
          <ul className="nav nav-tabs card-header-tabs">
              <li className="nav-item"><button className={resultStyle}
onClick={() => setTab(0)}>Result</button></li>
              <li className="nav-item"><button className={rawStyle} onClick=
{() => setTab(1)}>Raw Data</button></li>
          </ul>
        </div>
        {detailsBody}
      </div>
    </main>
  );
}
```

Next steps

- Index Azure SQL data

Create a search app in ASP.NET Core

Article • 04/09/2023 • 8 minutes to read

In this tutorial, create a basic ASP.NET Core (Model-View-Controller) app that runs in localhost and connects to the hotels-sample-index on your search service. In this tutorial, you'll learn to:

- ✓ Create a basic search page
- ✓ Filter results
- ✓ Sort results

This tutorial puts the focus on server-side operations called through the Search APIs. Although it's common to sort and filter in client-side script, knowing how to invoke these operations on the server gives you more options when designing the search experience.

Sample code for this tutorial can be found in the azure-search-dotnet-samples repository on GitHub.

Prerequisites

- Visual Studio
- Azure.Search.Documents NuGet package
- Azure Cognitive Search [1]
- Hotel samples index [2]

[1] The search service can be any tier, but it must have public network access for this tutorial.

[2] To complete this tutorial, you need to create the hotels-sample-index on your search service. Make sure the search index name is `hotels-sample-index`, or change the index name in the `HomeController.cs` file.

Create the project

1. Start Visual Studio and select **Create a new project**.

2. Select **ASP.NET Core Web App (Model-View-Controller)**, and then select **Next**.

3. Provide a project name, and then select **Next**.

4. On the next page, select **.NET 6.0** or **.NET 7.0**.

5. Verify that **Do not use top-level statements** is unchecked.

6. Select **Create**.

Add NuGet packages

1. On Tools, select **NuGet Package Manager** > **Manage NuGet Packages for the solution**.

2. Browse for `Azure.Search.Documents` and install the latest stable version.

3. Browse for and install the `Microsoft.Spatial` package. The sample index includes a GeographyPoint data type. Installing this package avoids run time errors. Alternatively, remove the "Location" field from the Hotels class if you don't want to install the package. It's not used in this tutorial.

Add service information

For the connection, the app presents a query API key to your fully qualified search URL. Both are specified in the `appsettings.json` file.

Modify `appsettings.json` to specify your search service and query API key.

JSON

```
{
    "SearchServiceUri": "<YOUR-SEARCH-SERVICE-URL>",
    "SearchServiceQueryApiKey": "<YOUR-SEARCH-SERVICE-QUERY-API-KEY>"
}
```

You can get the service URL and API key from the portal. Because this code is querying an index and not creating one, you can use a query key instead of an admin key.

Make sure to specify the search service that has the hotels-sample-index.

Add models

In this step, create models that represent the schema of the hotels-sample-index.

1. In Solution explorer, right-select **Models** and add a new class named "Hotel" for the following code:

   C#

```
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.Indexes;
using Microsoft.Spatial;
using System.Text.Json.Serialization;

namespace HotelDemoApp.Models
{
    public partial class Hotel
    {
        [SimpleField(IsFilterable = true, IsKey = true)]
        public string HotelId { get; set; }

        [SearchableField(IsSortable = true)]
        public string HotelName { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.EnLucene)]
        public string Description { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.FrLucene)]
        [JsonPropertyName("Description_fr")]
        public string DescriptionFr { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string Category { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string[] Tags { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public bool? ParkingIncluded { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public DateTimeOffset? LastRenovationDate { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public double? Rating { get; set; }

        public Address Address { get; set; }

        [SimpleField(IsFilterable = true, IsSortable = true)]
        public GeographyPoint Location { get; set; }

        public Rooms[] Rooms { get; set; }
    }
}
```

2. Add a class named "Address" and replace it with the following code:

C#

```
using Azure.Search.Documents.Indexes;

namespace HotelDemoApp.Models
{
    public partial class Address
    {
        [SearchableField]
        public string StreetAddress { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string City { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string StateProvince { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string PostalCode { get; set; }

        [SearchableField(IsFilterable = true, IsSortable = true,
IsFacetable = true)]
        public string Country { get; set; }
    }
}
```

3. Add a class named "Rooms" and replace it with the following code:

C#

```
using Azure.Search.Documents.Indexes.Models;
using Azure.Search.Documents.Indexes;
using System.Text.Json.Serialization;

namespace HotelDemoApp.Models
{
    public partial class Rooms
    {
        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.EnMicrosoft)]
        public string Description { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.FrMicrosoft)]
        [JsonPropertyName("Description_fr")]
        public string DescriptionFr { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string Type { get; set; }
```

```
        [SimpleField(IsFilterable = true, IsFacetable = true)]
        public double? BaseRate { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string BedOptions { get; set; }

        [SimpleField(IsFilterable = true, IsFacetable = true)]
        public int SleepsCount { get; set; }

        [SimpleField(IsFilterable = true, IsFacetable = true)]
        public bool? SmokingAllowed { get; set; }

        [SearchableField(IsFilterable = true, IsFacetable = true)]
        public string[] Tags { get; set; }
    }
}
```

4. Add a class named "SearchData" and replace it with the following code:

```
C#

using Azure.Search.Documents.Models;

namespace HotelDemoApp.Models
{
    public class SearchData
    {
        // The text to search for.
        public string searchText { get; set; }

        // The list of results.
        public SearchResults<Hotel> resultList;
    }
}
```

Modify the controller

For this tutorial, modify the default `HomeController` to contain methods that execute on your search service.

1. In Solution explorer under **Models**, open `HomeController`.

2. Replace the default with the following content:

```
C#

using Azure;
 using Azure.Search.Documents;
 using Azure.Search.Documents.Indexes;
```

```
using HotelDemoApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Diagnostics;

namespace HotelDemoApp.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        [HttpPost]
        public async Task<ActionResult> Index(SearchData model)
        {
            try
            {
                // Check for a search string
                if (model.searchText == null)
                {
                    model.searchText = "";
                }

                // Send the query to Search.
                await RunQueryAsync(model);
            }

            catch
            {
                return View("Error", new ErrorViewModel { RequestId =
"1" });
            }
            return View(model);
        }

        [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }

        private static SearchClient _searchClient;
        private static SearchIndexClient _indexClient;
        private static IConfigurationBuilder _builder;
        private static IConfigurationRoot _configuration;

        private void InitSearch()
        {
            // Create a configuration using appsettings.json
            _builder = new
ConfigurationBuilder().AddJsonFile("appsettings.json");
            _configuration = _builder.Build();
```

```
            // Read the values from appsettings.json
            string searchServiceUri =
_configuration["SearchServiceUri"];
            string queryApiKey =
_configuration["SearchServiceQueryApiKey"];

            // Create a service and index client.
            _indexClient = new SearchIndexClient(new
Uri(searchServiceUri), new AzureKeyCredential(queryApiKey));
            _searchClient = _indexClient.GetSearchClient("hotels-
sample-index");
        }

        private async Task<ActionResult> RunQueryAsync(SearchData
model)
        {
            InitSearch();

            var options = new SearchOptions()
            {
                IncludeTotalCount = true
            };

            // Enter Hotel property names to specify which fields are
returned.
            // If Select is empty, all "retrievable" fields are
returned.
            options.Select.Add("HotelName");
            options.Select.Add("Category");
            options.Select.Add("Rating");
            options.Select.Add("Tags");
            options.Select.Add("Address/City");
            options.Select.Add("Address/StateProvince");
            options.Select.Add("Description");

            // For efficiency, the search call should be asynchronous,
so use SearchAsync rather than Search.
            model.resultList = await _searchClient.SearchAsync<Hotel>
(model.searchText, options).ConfigureAwait(false);

            // Display the results.
            return View("Index", model);
        }
        public IActionResult Privacy()
        {
            return View();
        }
    }
 }
```

Modify the view

1. In Solution explorer under **Views** > **Home**, open `index.cshtml`.

2. Replace the default with the following content:

razor

```razor
@model HotelDemoApp.Models.SearchData;

@{
    ViewData["Title"] = "Index";
}

<div>
    <img src="~/images/azure-logo.png" width="80" />
    <h2>Search for Hotels</h2>

    <p>Use this demo app to test server-side sorting and filtering.
Modify the RunQueryAsync method to change the operation. The app uses
the default search configuration (simple search syntax, with
searchMode=Any).</p>

    <form asp-controller="Home" asp-action="Index">
        <p>
            <input type="text" name="searchText" />
            <input type="submit" value="Search" />
        </p>
    </form>
</div>

<div>
    @using (Html.BeginForm("Index", "Home", FormMethod.Post))
    {
        @if (Model != null)
        {
            // Show the result count.
            <p>@Model.resultList.TotalCount Results</p>

            // Get search results.
            var results = Model.resultList.GetResults().ToList();

            {
                <table class="table">
                    <thead>
                        <tr>
                            <th>Name</th>
                            <th>Category</th>
                            <th>Rating</th>
                            <th>Tags</th>
                            <th>City</th>
                            <th>State</th>
                            <th>Description</th>
                        </tr>
                    </thead>
                    <tbody>
```

```
                        @foreach (var d in results)
                        {
                            <tr>
                                <td>@d.Document.HotelName</td>
                                <td>@d.Document.Category</td>
                                <td>@d.Document.Rating</td>
                                <td>@d.Document.Tags[0]</td>
                                <td>@d.Document.Address.City</td>

<td>@d.Document.Address.StateProvince</td>
                                <td>@d.Document.Description</td>
                            </tr>
                        }
                    </tbody>
                </table>
            }
        }
    }
</div>
```

Run the sample

1. Press **F5** to compile and run the project. The app runs on local host and opens in your default browser.

2. Select **Search** to return all results.

3. This code uses the default search configuration, supporting the simple syntax and `searchMode=Any`. You can enter keywords, augment with Boolean operators, or run a prefix search (`pool*`).

In the next several sections, modify the **RunQueryAsync** method in the `HomeController` to add filters and sorting.

Filter results

Index field attributes determine which fields are searchable, filterable, sortable, facetable, and retrievable. In the hotels-sample-index, filterable fields include "Category", "Address/City", and "Address/StateProvince". This example adds a $Filter expression on "Category".

A filter always executes first, followed by a query assuming one is specified.

1. Open the `HomeController` and find the **RunQueryAsync** method. Add Filter to `var options = new SearchOptions()`:

```C#
private async Task<ActionResult> RunQueryAsync(SearchData model)
{
    InitSearch();

    var options = new SearchOptions()
    {
        IncludeTotalCount = true,
        Filter = "search.in(Category,'Budget,Suite')"
    };

    options.Select.Add("HotelName");
    options.Select.Add("Category");
    options.Select.Add("Rating");
    options.Select.Add("Tags");
    options.Select.Add("Address/City");
    options.Select.Add("Address/StateProvince");
    options.Select.Add("Description");

    model.resultList = await _searchClient.SearchAsync<Hotel>
(model.searchText, options).ConfigureAwait(false);

    return View("Index", model);
}
```

2. Run the application.

3. Select **Search** to run an empty query. The filter criteria returns 18 documents instead of the original 50.

For more information about filter expressions, see Filters in Azure Cognitive Search and OData $filter syntax in Azure Cognitive Search.

Sort results

In the hotels-sample-index, sortable fields include "Rating" and "LastRenovated". This example adds an $OrderBy expression to the "Rating" field.

1. Open the `HomeController` and replace **RunQueryAsync** method with the following version:

```C#
private async Task<ActionResult> RunQueryAsync(SearchData model)
{
    InitSearch();

    var options = new SearchOptions()
```

```
        {
            IncludeTotalCount = true,
        };

        options.OrderBy.Add("Rating desc");

        options.Select.Add("HotelName");
        options.Select.Add("Category");
        options.Select.Add("Rating");
        options.Select.Add("Tags");
        options.Select.Add("Address/City");
        options.Select.Add("Address/StateProvince");
        options.Select.Add("Description");

        model.resultList = await _searchClient.SearchAsync<Hotel>
(model.searchText, options).ConfigureAwait(false);

        return View("Index", model);
    }
```

2. Run the application. Results are sorted by "Rating" in descending order.

For more information about sorting, see OData $orderby syntax in Azure Cognitive Search.

Next steps

In this tutorial, you created an ASP.NET Core (MVC) project that connected to a search service and called Search APIs for server-side filtering and sorting.

If you want to explore client-side code that respond to user actions, consider adding a React template to your solution:

C# Tutorial: Add search to a website with .NET

Tutorial: Index Azure SQL data using the .NET SDK

Article • 10/04/2022 • 9 minutes to read

Configure an indexer to extract searchable data from Azure SQL Database, sending it to a search index in Azure Cognitive Search.

This tutorial uses C# and the .NET SDK to perform the following tasks:

- ✓ Create a data source that connects to Azure SQL Database
- ✓ Create an indexer
- ✓ Run an indexer to load data into an index
- ✓ Query an index as a verification step

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- Azure SQL Database
- Visual Studio
- Create or find an existing search service

> **Note**
>
> You can use the free service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

Download files

Source code for this tutorial is in the DotNetHowToIndexer folder in the Azure-Samples/search-dotnet-getting-started GitHub repository.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, and Azure SQL Database as an external data source. If possible, create both services in the same region

and resource group for proximity and manageability. In practice, Azure SQL Database can be in any region.

Start with Azure SQL Database

In this step, create an external data source on Azure SQL Database that an indexer can crawl. You can use the Azure portal and the *hotels.sql* file from the sample download to create the dataset in Azure SQL Database. Azure Cognitive Search consumes flattened rowsets, such as one generated from a view or query. The SQL file in the sample solution creates and populates a single table.

If you have an existing Azure SQL Database resource, you can add the hotels table to it, starting at step 4.

1. Sign in to the Azure portal.

2. Find or create a **SQL Database**. You can use defaults and the lowest level pricing tier. One advantage to creating a server is that you can specify an administrator user name and password, necessary for creating and loading tables in a later step.

3. Select **Review + create** to deploy the new server and database. Wait for the server and database to deploy. Go to the resource.

4. On the navigate pane, select **Getting started** and then select **Configure** to allow access.

5. Under Public access, click **Selected networks**.

6. Under Firewall rules, add your client IPv4 address. This is the portal client.

7. Under Exception, select **Allow Azure services and resources to access this server**.

8. Save your changes and then close the Networking page.

9. On the navigation pane, select **Query editor (preview)** and enter the user name and password of server admin.

   You'll probably get an access denied error. Copy the client IP address from the error message. Return to the firewall rules page to add a rule that allows access

from your client.

10. In Query editor, select **Open query** and navigate to the location of *hotels.sql* file on your local computer.

11. Select the file and select **Open**. The script should look similar to the following screenshot:

12. Select **Run** to execute the query. In the Results pane, you should see a query succeeded message, for three rows.

13. To return a rowset from this table, you can execute the following query as a verification step:

```
SQL

SELECT * FROM Hotels
```

14. Copy the ADO.NET connection string for the database. Under **Settings** > **Connection Strings**, copy the ADO.NET connection string, similar to the example below.

```
SQL
```

```
Server=tcp:{your_dbname}.database.windows.net,1433;Initial
Catalog=hotels-db;Persist Security Info=False;User ID=
{your_username};Password=
{your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServer
Certificate=False;Connection Timeout=30;
```

You'll need this connection string in the next exercise, setting up your environment.

Azure Cognitive Search

The next component is Azure Cognitive Search, which you can create in the portal. You can use the Free tier to complete this walkthrough.

Get an admin api-key and URL for Azure Cognitive Search

API calls require the service URL and an access key. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

2 - Set up your environment

1. Start Visual Studio and open **DotNetHowToIndexers.sln**.

2. In Solution Explorer, open **appsettings.json** to provide connection information.

3. For `SearchServiceEndPoint`, if the full URL on the service overview page is "https://my-demo-service.search.windows.net", then the value to provide is the entire URL.

4. For `AzureSqlConnectionString`, the string format is similar to this: `"Server=tcp:{your_dbname}.database.windows.net,1433;Initial Catalog=hotels-db;Persist Security Info=False;User ID={your_username};Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"`

   JSON

   ```
   {
     "SearchServiceEndPoint": "<placeholder-search-full-url>",
     "SearchServiceAdminApiKey": "<placeholder-admin-key-for-search-service>",
     "AzureSqlConnectionString": "<placeholder-ADO.NET-connection-string",
   }
   ```

5. Replace the user password in the SQL connection string to a valid password. While the database and user names will copy over, the password must be entered manually.

3 - Create the pipeline

Indexers require a data source object and an index. Relevant code is in two files:

- **hotel.cs**, containing a schema that defines the index
- **Program.cs**, containing functions for creating and managing structures in your service

In hotel.cs

The index schema defines the fields collection, including attributes specifying allowed operations, such as whether a field is full-text searchable, filterable, or sortable as shown

in the following field definition for HotelName. A SearchableField is full-text searchable by definition. Other attributes are assigned explicitly.

C#

```
. . .
[SearchableField(IsFilterable = true, IsSortable = true)]
[JsonPropertyName("hotelName")]
public string HotelName { get; set; }
. . .
```

A schema can also include other elements, including scoring profiles for boosting a search score, custom analyzers, and other constructs. However, for our purposes, the schema is sparsely defined, consisting only of fields found in the sample datasets.

In Program.cs

The main program includes logic for creating an indexer client, an index, a data source, and an indexer. The code checks for and deletes existing resources of the same name, under the assumption that you might run this program multiple times.

The data source object is configured with settings that are specific to Azure SQL Database resources, including partial or incremental indexing for using the built-in change detection features of Azure SQL. The source demo hotels database in Azure SQL has a "soft delete" column named **IsDeleted**. When this column is set to true in the database, the indexer removes the corresponding document from the Azure Cognitive Search index.

C#

```
Console.WriteLine("Creating data source...");

var dataSource =
      new SearchIndexerDataSourceConnection(
         "hotels-sql-ds",
         SearchIndexerDataSourceType.AzureSql,
         configuration["AzureSQLConnectionString"],
         new SearchIndexerDataContainer("hotels"));

indexerClient.CreateOrUpdateDataSourceConnection(dataSource);
```

An indexer object is platform-agnostic, where configuration, scheduling, and invocation are the same regardless of the source. This example indexer includes a schedule, a reset option that clears indexer history, and calls a method to create and run the indexer immediately. To create or update an indexer, use CreateOrUpdateIndexerAsync.

C#

```
Console.WriteLine("Creating Azure SQL indexer...");

var schedule = new IndexingSchedule(TimeSpan.FromDays(1))
{
      StartTime = DateTimeOffset.Now
};

var parameters = new IndexingParameters()
{
      BatchSize = 100,
      MaxFailedItems = 0,
      MaxFailedItemsPerBatch = 0
};

// Indexer declarations require a data source and search index.
// Common optional properties include a schedule, parameters, and field
mappings
// The field mappings below are redundant due to how the Hotel class is
defined, but
// we included them anyway to show the syntax
var indexer = new SearchIndexer("hotels-sql-idxr", dataSource.Name,
searchIndex.Name)
{
      Description = "Data indexer",
      Schedule = schedule,
      Parameters = parameters,
      FieldMappings =
      {
         new FieldMapping("_id") {TargetFieldName = "HotelId"},
         new FieldMapping("Amenities") {TargetFieldName = "Tags"}
      }
};

await indexerClient.CreateOrUpdateIndexerAsync(indexer);
```

Indexer runs are usually scheduled, but during development you might want to run the indexer immediately using RunIndexerAsync.

C#

```
Console.WriteLine("Running Azure SQL indexer...");

try
{
      await indexerClient.RunIndexerAsync(indexer.Name);
}
catch (CloudException e) when (e.Response.StatusCode == (HttpStatusCode)429)
{
      Console.WriteLine("Failed to run indexer: {0}", e.Response.Content);
}
```

4 - Build the solution

Press F5 to build and run the solution. The program executes in debug mode. A console window reports the status of each operation.

Your code runs locally in Visual Studio, connecting to your search service on Azure, which in turn connects to Azure SQL Database and retrieves the dataset. With this many operations, there are several potential points of failure. If you get an error, check the following conditions first:

- Search service connection information that you provide is the full URL. If you entered just the service name, operations stop at index creation, with a failure to connect error.
- Database connection information in **appsettings.json**. It should be the ADO.NET connection string obtained from the portal, modified to include a username and password that are valid for your database. The user account must have permission to retrieve data. Your local client IP address must be allowed inbound access through the firewall.
- Resource limits. Recall that the Free tier has limits of three indexes, indexers, and data sources. A service at the maximum limit can't create new objects.

5 - Search

Use Azure portal to verify object creation, and then use **Search explorer** to query the index.

1. Sign in to the Azure portal, and in your search service **Overview** page, open each list in turn to verify the object is created. **Indexes**, **Indexers**, and **Data Sources** will have "hotels", "azure-sql-indexer", and "azure-sql", respectively.

2. On the Indexes tab, select the hotels index. On the hotels page, **Search explorer** is the first tab.

3. Select **Search** to issue an empty query.

   The three entries in your index are returned as JSON documents. Search explorer returns documents in JSON so that you can view the entire structure.

4. Next, enter a search string: `search=river&$count=true`.

   This query invokes full text search on the term `river`, and the result includes a count of the matching documents. Returning the count of matching documents is helpful in testing scenarios when you have a large index with thousands or millions of documents. In this case, only one document matches the query.

5. Lastly, enter a search string that limits the JSON output to fields of interest: `search=river&$count=true&$select=hotelId, baseRate, description`.

   The query response is reduced to selected fields, resulting in more concise output.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing objects and deletes them so that you can rerun your code.

You can also use the portal to delete indexes, indexers, and data sources.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with the basics of SQL Database indexing, let's take a closer look at indexer configuration.

Configure a SQL Database indexer

Tutorial: Index JSON blobs from Azure Storage using REST

Article • 03/24/2023 • 9 minutes to read

Azure Cognitive Search can index JSON documents and arrays in Azure Blob Storage using an indexer that knows how to read semi-structured data. Semi-structured data contains tags or markings which separate content within the data. It splits the difference between unstructured data, which must be fully indexed, and formally structured data that adheres to a data model, such as a relational database schema, that can be indexed on a per-field basis.

This tutorial uses Postman and the Search REST APIs to perform the following tasks:

- ✓ Configure an Azure Cognitive Search data source for an Azure blob container
- ✓ Create an Azure Cognitive Search index to contain searchable content
- ✓ Configure and run an indexer to read the container and extract searchable content from Azure Blob Storage
- ✓ Search the index you just created

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

- Azure Storage
- Postman app
- Create or find an existing search service

> **Note**
>
> You can use the free service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

Download files

Clinical-trials-json.zip contains the data used in this tutorial. Download and unzip this file to its own folder. Data originates from clinicaltrials.gov, converted to JSON for this tutorial.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, and Azure Blob Storage to provide the data.

If possible, create both in the same region and resource group for proximity and manageability. In practice, your Azure Storage account can be in any region.

Start with Azure Storage

1. Sign in to the Azure portal and click + **Create Resource**.

2. Search for *storage account* and select Microsoft's Storage Account offering.

3. In the Basics tab, the following items are required. Accept the defaults for everything else.

    - **Resource group**. Select an existing one or create a new one, but use the same group for all services so that you can manage them collectively.

    - **Storage account name**. If you think you might have multiple resources of the same type, use the name to disambiguate by type and region, for example *blobstoragewestus*.

    - **Location**. If possible, choose the same location used for Azure Cognitive Search and Cognitive Services. A single location voids bandwidth charges.

    - **Account Kind**. Choose the default, *StorageV2 (general purpose v2)*.

4. Click **Review + Create** to create the service.

5. Once it's created, click **Go to the resource** to open the Overview page.

6. Click **Blobs** service.

7. Create a Blob container to contain sample data. You can set the Public Access Level to any of its valid values.

8. After the container is created, open it and select **Upload** on the command bar.

9. Navigate to the folder containing the sample files. Select all of them and then click **Upload**.

After the upload completes, the files should appear in their own subfolder inside the data container.

Azure Cognitive Search

The next resource is Azure Cognitive Search, which you can create in the portal. You can use the Free tier to complete this walkthrough.

As with Azure Blob Storage, take a moment to collect the access key. Further on, when you begin structuring requests, you will need to provide the endpoint and admin api-key used to authenticate each request.

Get a key and URL

REST calls require the service URL and an access key on every request. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Set up Postman

Start Postman and set up an HTTP request. If you are unfamiliar with this tool, see Create a search index using REST APIs.

The request methods for every call in this tutorial are **POST** and **GET**. You'll make three API calls to your search service to create a data source, an index, and an indexer. The data source includes a pointer to your storage account and your JSON data. Your search service makes the connection when loading the data.

In Headers, set "Content-type" to `application/json` and set `api-key` to the admin api-key of your Azure Cognitive Search service. Once you set the headers, you can use them for every request in this exercise.

URIs must specify an api-version and each call should return a **201 Created**. The generally available api-version for using JSON arrays is `2020-06-30`.

3 - Create a data source

The Create Data Source API creates an Azure Cognitive Search object that specifies what data to index.

1. Set the endpoint of this call to `https://[service name].search.windows.net/datasources?api-version=2020-06-30`. Replace `[service name]` with the name of your search service.

2. Copy the following JSON into the request body.

```JSON
{
    "name" : "clinical-trials-json-ds",
    "type" : "azureblob",
    "credentials" : { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=[storage account
name];AccountKey=[storage account key];" },
    "container" : { "name" : "[blob container name]"}
}
```

3. Replace the connection string with a valid string for your account.

4. Replace "[blob container name]" with the container you created for the sample data.

5. Send the request. The response should look like:

```JSON
{
    "@odata.context":
"https://exampleurl.search.windows.net/$metadata#datasources/$entity",
    "@odata.etag": "\"0x8D505FBC3856C9E\"",
```

```
    "name": "clinical-trials-json-ds",
    "description": null,
    "type": "azureblob",
    "subtype": null,
    "credentials": {
        "connectionString":
"DefaultEndpointsProtocol=https;AccountName=
[mystorageaccounthere];AccountKey=[[myaccountkeyhere]]];"
    },
    "container": {
        "name": "[mycontainernamehere]",
        "query": null
    },
    "dataChangeDetectionPolicy": null,
    "dataDeletionDetectionPolicy": null
}
```

4 - Create an index

The second call is Create Index API, creating an Azure Cognitive Search index that stores all searchable data. An index specifies all the parameters and their attributes.

1. Set the endpoint of this call to `https://[service name].search.windows.net/indexes?api-version=2020-06-30`. Replace `[service name]` with the name of your search service.

2. Copy the following JSON into the request body.

JSON

```
{
  "name": "clinical-trials-json-index",
  "fields": [
  {"name": "FileName", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": false, "filterable": false,
"sortable": true},
  {"name": "Description", "type": "Edm.String", "searchable": true,
"retrievable": false, "facetable": false, "filterable": false,
"sortable": false},
  {"name": "MinimumAge", "type": "Edm.Int32", "searchable": false,
"retrievable": true, "facetable": true, "filterable": true, "sortable":
true},
  {"name": "Title", "type": "Edm.String", "searchable": true,
"retrievable": true, "facetable": false, "filterable": true,
"sortable": true},
  {"name": "URL", "type": "Edm.String", "searchable": false,
"retrievable": false, "facetable": false, "filterable": false,
"sortable": false},
  {"name": "MyURL", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": false, "filterable": false,
```

```
"sortable": false},
  {"name": "Gender", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": true, "filterable": true, "sortable":
false},
  {"name": "MaximumAge", "type": "Edm.Int32", "searchable": false,
"retrievable": true, "facetable": true, "filterable": true, "sortable":
true},
  {"name": "Summary", "type": "Edm.String", "searchable": true,
"retrievable": true, "facetable": false, "filterable": false,
"sortable": false},
  {"name": "NCTID", "type": "Edm.String", "key": true, "searchable":
true, "retrievable": true, "facetable": false, "filterable": true,
"sortable": true},
  {"name": "Phase", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": true, "filterable": true, "sortable":
false},
  {"name": "Date", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": false, "filterable": false,
"sortable": true},
  {"name": "OverallStatus", "type": "Edm.String", "searchable": false,
"retrievable": true, "facetable": true, "filterable": true, "sortable":
false},
  {"name": "OrgStudyId", "type": "Edm.String", "searchable": true,
"retrievable": true, "facetable": false, "filterable": true,
"sortable": false},
  {"name": "HealthyVolunteers", "type": "Edm.String", "searchable":
false, "retrievable": true, "facetable": true, "filterable": true,
"sortable": false},
  {"name": "Keywords", "type": "Collection(Edm.String)", "searchable":
true, "retrievable": true, "facetable": true, "filterable": false,
"sortable": false},
  {"name": "metadata_storage_last_modified",
"type":"Edm.DateTimeOffset", "searchable": false, "retrievable": true,
"filterable": true, "sortable": false},
  {"name": "metadata_storage_size", "type":"Edm.String", "searchable":
false, "retrievable": true, "filterable": true, "sortable": false},
  {"name": "metadata_content_type", "type":"Edm.String", "searchable":
true, "retrievable": true, "filterable": true, "sortable": false}
  ]
}
```

3. Send the request. The response should look like:

JSON

```
{
    "@odata.context":
"https://exampleurl.search.windows.net/$metadata#indexes/$entity",
    "@odata.etag": "\"0x8D505FC00EDD5FA\"",
    "name": "clinical-trials-json-index",
    "fields": [
        {
            "name": "FileName",
```

```
            "type": "Edm.String",
            "searchable": false,
            "filterable": false,
            "retrievable": true,
            "sortable": true,
            "facetable": false,
            "key": false,
            "indexAnalyzer": null,
            "searchAnalyzer": null,
            "analyzer": null,
            "synonymMaps": []
        },
        {
            "name": "Description",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "retrievable": false,
            "sortable": false,
            "facetable": false,
            "key": false,
            "indexAnalyzer": null,
            "searchAnalyzer": null,
            "analyzer": null,
            "synonymMaps": []
        },
        ...
    }
```

5 - Create and run an indexer

An indexer connects to the data source, imports data into the target search index, and optionally provides a schedule to automate the data refresh. The REST API is Create Indexer.

1. Set the URI for this call to `https://[service name].search.windows.net/indexers?api-version=2020-06-30`. Replace `[service name]` with the name of your search service.

2. Copy the following JSON into the request body.

JSON

```
{
  "name" : "clinical-trials-json-indexer",
  "dataSourceName" : "clinical-trials-json-ds",
  "targetIndexName" : "clinical-trials-json-index",
  "parameters" : { "configuration" : { "parsingMode" : "jsonArray" } }
}
```

3. Send the request. The request is processed immediately. When the response comes back, you will have an index that is full-text searchable. The response should look like:

JSON

```
{
    "@odata.context":
"https://exampleurl.search.windows.net/$metadata#indexers/$entity",
    "@odata.etag": "\"0x8D505FDE143D164\"",
    "name": "clinical-trials-json-indexer",
    "description": null,
    "dataSourceName": "clinical-trials-json-ds",
    "targetIndexName": "clinical-trials-json-index",
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "maxFailedItemsPerBatch": null,
        "base64EncodeKeys": null,
        "configuration": {
            "parsingMode": "jsonArray"
        }
    },
    "fieldMappings": [],
    "enrichers": [],
    "disabled": null
}
```

6 - Search your JSON files

You can start searching as soon as the first document is loaded.

1. Change the verb to **GET**.

2. Set the URI for this call to `https://[service name].search.windows.net/indexes/clinical-trials-json-index/docs?search=*&api-version=2020-06-30&$count=true`. Replace `[service name]` with the name of your search service.

3. Send the request. This is an unspecified full text search query that returns all of the fields marked as retrievable in the index, along with a document count. The response should look like:

JSON

```
{
    "@odata.context":
"https://exampleurl.search.windows.net/indexes('clinical-trials-json-
index')/$metadata#docs(*)",
    "@odata.count": 100,
    "value": [
        {
            "@search.score": 1.0,
            "FileName": "NCT00000102.txt",
            "MinimumAge": 14,
            "Title": "Congenital Adrenal Hyperplasia: Calcium Channels
as Therapeutic Targets",
            "MyURL": "https://azure.storagedemos.com/clinical-
trials/NCT00000102.txt",
            "Gender": "Both",
            "MaximumAge": 35,
            "Summary": "This study will test the ability of extended
release nifedipine (Procardia XL), a blood pressure medication, to
permit a decrease in the dose of glucocorticoid medication children
take to treat congenital adrenal hyperplasia (CAH).",
            "NCTID": "NCT00000102",
            "Phase": "Phase 1/Phase 2",
            "Date": "ClinicalTrials.gov processed this data on October
25, 2016",
            "OverallStatus": "Completed",
            "OrgStudyId": "NCRR-M01RR01070-0506",
            "HealthyVolunteers": "No",
            "Keywords": [],
            "metadata_storage_last_modified": "2019-04-09T18:16:24Z",
            "metadata_storage_size": "33060",
            "metadata_content_type": null
        },
        . . .
```


4. Add the `$select` query parameter to limit the results to fewer fields: `https://[service name].search.windows.net/indexes/clinical-trials-json-index/docs?search=*&$select=Gender,metadata_storage_size&api-version=2020-06-30&$count=true`. For this query, 100 documents match, but by default, Azure Cognitive Search only returns 50 in the results.

5. An example of more complex query would include `$filter=MinimumAge ge 30 and MaximumAge lt 75`, which returns only results where the parameters MinimumAge is greater than or equal to 30 and MaximumAge is less than 75. Replace the `$select` expression with the `$filter` expression.

You can also use Logical operators (and, or, not) and comparison operators (eq, ne, gt, lt, ge, le). String comparisons are case-sensitive. For more information and examples, see Create a simple query.

> **Note**
>
> The `$filter` parameter only works with metadata that were marked filterable at the creation of your index.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

You can use the portal to delete indexes, indexers, and data sources. Or use **DELETE** and provide URLs to each object. The following command deletes an indexer.

HTTP

```
DELETE https://[YOUR-SERVICE-NAME].search.windows.net/indexers/clinical-trials-json-indexer?api-version=2020-06-30
```

Status code 204 is returned on successful deletion.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with the basics of Azure Blob indexing, let's take a closer look at indexer configuration for JSON blobs in Azure Storage.

Configure JSON blob indexing

Tutorial: Index from multiple data sources using the .NET SDK

Article • 03/10/2023 • 14 minutes to read

Azure Cognitive Search can import, analyze, and index data from multiple data sources into a single consolidated search index.

This tutorial uses C# and the Azure.Search.Documents client library in the Azure SDK for .NET to index sample hotel data from an Azure Cosmos DB instance, and merge that with hotel room details drawn from Azure Blob Storage documents. The result will be a combined hotel search index containing hotel documents, with rooms as a complex data types.

In this tutorial, you'll perform the following tasks:

- ✓ Upload sample data and create data sources
- ✓ Identify the document key
- ✓ Define and create the index
- ✓ Index hotel data from Azure Cosmos DB
- ✓ Merge hotel room data from blob storage

If you don't have an Azure subscription, create a free account before you begin.

Overview

This tutorial uses Azure.Search.Documents to create and run multiple indexers. In this tutorial, you'll set up two Azure data sources so that you can configure an indexer that pulls from both to populate a single search index. The two data sets must have a value in common to support the merge. In this sample, that field is an ID. As long as there's a field in common to support the mapping, an indexer can merge data from disparate resources: structured data from Azure SQL, unstructured data from Blob storage, or any combination of supported data sources on Azure.

A finished version of the code in this tutorial can be found in the following project:

- multiple-data-sources/v11 (GitHub)

For an earlier version of the .NET SDK, see Microsoft.Azure.Search (version 10) code sample on GitHub.

Prerequisites

- Azure Cosmos DB
- Azure Storage
- Visual Studio
- Azure Cognitive Search (version 11.x) NuGet package
- Azure Cognitive Search

> **Note**
>
> You can use the free service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, Azure Cosmos DB for one data set, and Azure Blob Storage for the second data set.

If possible, create all services in the same region and resource group for proximity and manageability. In practice, your services can be in any region.

This sample uses two small sets of data that describe seven fictional hotels. One set describes the hotels themselves, and will be loaded into an Azure Cosmos DB database. The other set contains hotel room details, and is provided as seven separate JSON files to be uploaded into Azure Blob Storage.

Start with Azure Cosmos DB

1. Sign in to the Azure portal, and then navigate your Azure Cosmos DB account Overview page.

2. Select **Data Explorer** and then select **New Database**.

3. Enter the name **hotel-rooms-db**. Accept default values for the remaining settings.

4. Create a new container. Use the existing database you just created. Enter **hotels** for the container name, and use **/HotelId** for the Partition key.

5. Select **Items** under **hotels**, and then select **Upload Item** on the command bar. Navigate to and then select the file **cosmosdb/HotelsDataSubset_CosmosDb.json** in the project folder.

6. Use the Refresh button to refresh your view of the items in the hotels collection. You should see seven new database documents listed.

7. Copy a connection string from the **Keys** page into Notepad. You'll need this value for **appsettings.json** in a later step. If you didn't use the suggested database name "hotel-rooms-db", copy the database name as well.

Azure Blob Storage

1. Sign in to the Azure portal, navigate to your Azure storage account, select **Blobs**, and then select **+ Container**.

2. Create a blob container named **hotel-rooms** to store the sample hotel room JSON files. You can set the Public Access Level to any of its valid values.

3. After the container is created, open it and select **Upload** on the command bar. Navigate to the folder containing the sample files. Select all of them and then select **Upload**.

4. Copy the storage account name and a connection string from the **Access Keys** page into Notepad. You'll need both values for **appsettings.json** in a later step.

Azure Cognitive Search

The third component is Azure Cognitive Search, which you can create in the portal or find an existing search service in your Azure resources.

Copy an admin api-key and URL for Azure Cognitive Search

To authenticate to your search service, you'll need the service URL and an access key.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Set up your environment

1. Start Visual Studio and in the **Tools** menu, select **NuGet Package Manager** and then **Manage NuGet Packages for Solution...**.

2. In the **Browse** tab, find and then install **Azure.Search.Documents** (version 11.0 or later).

3. Search for the **Microsoft.Extensions.Configuration** and **Microsoft.Extensions.Configuration.Json** NuGet packages and install them as well.

4. Open the solution file **/v11/AzureSearchMultipleDataSources.sln**.

5. In Solution Explorer, edit the **appsettings.json** file to add connection information.

JSON

```
{
  "SearchServiceUri": "<YourSearchServiceURL>",
  "SearchServiceAdminApiKey": "<YourSearchServiceAdminApiKey>",
  "BlobStorageAccountName": "<YourBlobStorageAccountName>",
  "BlobStorageConnectionString": "<YourBlobStorageConnectionString>",
  "CosmosDBConnectionString": "<YourCosmosDBConnectionString>",
  "CosmosDBDatabaseName": "hotel-rooms-db"
}
```

The first two entries are the URL and admin keys of a search service. Use the full endpoint, for example: `https://mydemo.search.windows.net`.

The next entries specify account names and connection string information for the Azure Blob Storage and Azure Cosmos DB data sources.

3 - Map key fields

Merging content requires that both data streams are targeting the same documents in the search index.

In Azure Cognitive Search, the key field uniquely identifies each document. Every search index must have exactly one key field of type `Edm.String`. That key field must be present for each document in a data source that is added to the index. (In fact, it's the only required field.)

When indexing data from multiple data sources, make sure each incoming row or document contains a common document key to merge data from two physically distinct source documents into a new search document in the combined index.

It often requires some up-front planning to identify a meaningful document key for your index, and make sure it exists in both data sources. In this demo, the `HotelId` key for each hotel in Azure Cosmos DB is also present in the rooms JSON blobs in Blob storage.

Azure Cognitive Search indexers can use field mappings to rename and even reformat data fields during the indexing process, so that source data can be directed to the correct index field. For example, in Azure Cosmos DB, the hotel identifier is called **`HotelId`**. But in the JSON blob files for the hotel rooms, the hotel identifier is named **`Id`**. The program handles this discrepancy by mapping the **`Id`** field from the blobs to the **`HotelId`** key field in the indexer.

> **Note**
>
> In most cases, auto-generated document keys, such as those created by default by some indexers, do not make good document keys for combined indexes. In general you will want to use a meaningful, unique key value that already exists in, or can be easily added to, your data sources.

4 - Explore the code

Once the data and configuration settings are in place, the sample program in **/v11/AzureSearchMultipleDataSources.sln** should be ready to build and run.

This simple C#/.NET console app performs the following tasks:

- Creates a new index based on the data structure of the C# Hotel class (which also references the Address and Room classes).
- Creates a new data source and an indexer that maps Azure Cosmos DB data to index fields. These are both objects in Azure Cognitive Search.
- Runs the indexer to load Hotel data from Azure Cosmos DB.
- Creates a second data source and an indexer that maps JSON blob data to index fields.
- Runs the second indexer to load Rooms data from Blob storage.

Before running the program, take a minute to study the code and the index and indexer definitions for this sample. The relevant code is in two files:

- **Hotel.cs** contains the schema that defines the index
- **Program.cs** contains functions that create the Azure Cognitive Search index, data sources, and indexers, and load the combined results into the index.

Create an index

This sample program uses CreateIndexAsync to define and create an Azure Cognitive Search index. It takes advantage of the FieldBuilder class to generate an index structure from a C# data model class.

The data model is defined by the Hotel class, which also contains references to the Address and Room classes. The FieldBuilder drills down through multiple class definitions to generate a complex data structure for the index. Metadata tags are used to define the attributes of each field, such as whether it's searchable or sortable.

The program will delete any existing index of the same name before creating the new one, in case you want to run this example more than once.

The following snippets from the Hotel.cs file show single fields, followed by a reference to another data model class, Room[], which in turn is defined in **Room.cs** file (not shown).

C#

```
. . .
[SimpleField(IsFilterable = true, IsKey = true)]
public string HotelId { get; set; }
```

```
[SearchableField(IsFilterable = true, IsSortable = true)]
public string HotelName { get; set; }
. . .
public Room[] Rooms { get; set; }
. . .
```

In the **Program.cs** file, a SearchIndex is defined with a name and a field collection generated by the `FieldBuilder.Build` method, and then created as follows:

```
C#

private static async Task CreateIndexAsync(string indexName,
SearchIndexClient indexClient)
{
    // Create a new search index structure that matches the properties of
the Hotel class.
    // The Address and Room classes are referenced from the Hotel class. The
FieldBuilder
    // will enumerate these to create a complex data structure for the
index.
    FieldBuilder builder = new FieldBuilder();
    var definition = new SearchIndex(indexName,
builder.Build(typeof(Hotel)));

    await indexClient.CreateIndexAsync(definition);
}
```

Create Azure Cosmos DB data source and indexer

Next the main program includes logic to create the Azure Cosmos DB data source for the hotels data.

First it concatenates the Azure Cosmos DB database name to the connection string. Then it defines a SearchIndexerDataSourceConnection object.

```
C#

private static async Task CreateAndRunCosmosDbIndexerAsync(string indexName,
SearchIndexerClient indexerClient)
{
    // Append the database name to the connection string
    string cosmosConnectString =
        configuration["CosmosDBConnectionString"]
        + ";Database="
        + configuration["CosmosDBDatabaseName"];

    SearchIndexerDataSourceConnection cosmosDbDataSource = new
SearchIndexerDataSourceConnection(
        name: configuration["CosmosDBDatabaseName"],
```

```
        type: SearchIndexerDataSourceType.CosmosDb,
        connectionString: cosmosConnectString,
        container: new SearchIndexerDataContainer("hotels"));

    // The Azure Cosmos DB data source does not need to be deleted if it
already exists,
    // but the connection string might need to be updated if it has changed.
    await
indexerClient.CreateOrUpdateDataSourceConnectionAsync(cosmosDbDataSource);
```

After the data source is created, the program sets up an Azure Cosmos DB indexer named **hotel-rooms-cosmos-indexer**.

The program will update any existing indexers with the same name, overwriting the existing indexer with the content of the above code. It also includes reset and run actions, in case you want to run this example more than once.

The following example defines a schedule for the indexer, so that it will run once per day. You can remove the schedule property from this call if you don't want the indexer to automatically run again in the future.

C#

```
SearchIndexer cosmosDbIndexer = new SearchIndexer(
    name: "hotel-rooms-cosmos-indexer",
    dataSourceName: cosmosDbDataSource.Name,
    targetIndexName: indexName)
{
    Schedule = new IndexingSchedule(TimeSpan.FromDays(1))
};

// Indexers keep metadata about how much they have already indexed.
// If we already ran the indexer, it "remembers" and does not run again.
// To avoid this, reset the indexer if it exists.
try
{
    await indexerClient.GetIndexerAsync(cosmosDbIndexer.Name);
    // Reset the indexer if it exists.
    await indexerClient.ResetIndexerAsync(cosmosDbIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    // If the indexer does not exist, 404 will be thrown.
}

await indexerClient.CreateOrUpdateIndexerAsync(cosmosDbIndexer);

Console.WriteLine("Running Azure Cosmos DB indexer...\n");

try
{
```

```
    // Run the indexer.
    await indexerClient.RunIndexerAsync(cosmosDbIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 429)
{
    Console.WriteLine("Failed to run indexer: {0}", ex.Message);
}
```

This example includes a simple try-catch block to report any errors that might occur during execution.

After the Azure Cosmos DB indexer has run, the search index will contain a full set of sample hotel documents. However the rooms field for each hotel will be an empty array, since the Azure Cosmos DB data source omits room details. Next, the program will pull from Blob storage to load and merge the room data.

Create Blob storage data source and indexer

To get the room details, the program first sets up a Blob storage data source to reference a set of individual JSON blob files.

C#

```
private static async Task CreateAndRunBlobIndexerAsync(string indexName,
SearchIndexerClient indexerClient)
{
    SearchIndexerDataSourceConnection blobDataSource = new
SearchIndexerDataSourceConnection(
        name: configuration["BlobStorageAccountName"],
        type: SearchIndexerDataSourceType.AzureBlob,
        connectionString: configuration["BlobStorageConnectionString"],
        container: new SearchIndexerDataContainer("hotel-rooms"));

    // The blob data source does not need to be deleted if it already
exists,
    // but the connection string might need to be updated if it has changed.
    await
indexerClient.CreateOrUpdateDataSourceConnectionAsync(blobDataSource);
```

After the data source is created, the program sets up a blob indexer named **hotel-rooms-blob-indexer**, as shown below.

The JSON blobs contain a key field named `Id` instead of `HotelId`. The code uses the `FieldMapping` class to tell the indexer to direct the `Id` field value to the `HotelId` document key in the index.

Blob storage indexers can use IndexingParameters to specify a parsing mode. You should set different parsing modes depending on whether blobs represent a single document, or multiple documents within the same blob. In this example, each blob represents a single JSON document, so the code uses the `json` parsing mode. For more information about indexer parsing parameters for JSON blobs, see Index JSON blobs.

This example defines a schedule for the indexer, so that it will run once per day. You can remove the schedule property from this call if you don't want the indexer to automatically run again in the future.

C#

```
IndexingParameters parameters = new IndexingParameters();
parameters.Configuration.Add("parsingMode", "json");

SearchIndexer blobIndexer = new SearchIndexer(
    name: "hotel-rooms-blob-indexer",
    dataSourceName: blobDataSource.Name,
    targetIndexName: indexName)
{
    Parameters = parameters,
    Schedule = new IndexingSchedule(TimeSpan.FromDays(1))
};

// Map the Id field in the Room documents to the HotelId key field in the
index
blobIndexer.FieldMappings.Add(new FieldMapping("Id") { TargetFieldName =
"HotelId" });

// Reset the indexer if it already exists
try
{
    await indexerClient.GetIndexerAsync(blobIndexer.Name);
    await indexerClient.ResetIndexerAsync(blobIndexer.Name);
}
catch (RequestFailedException ex) when (ex.Status == 404) { }

await indexerClient.CreateOrUpdateIndexerAsync(blobIndexer);

try
{
    // Run the indexer.
    await searchService.Indexers.RunAsync(blobIndexer.Name);
}
catch (CloudException e) when (e.Response.StatusCode == (HttpStatusCode)429)
{
    Console.WriteLine("Failed to run indexer: {0}", e.Response.Content);
}
```

Because the index has already been populated with hotel data from the Azure Cosmos DB database, the blob indexer updates the existing documents in the index and adds the room details.

> ⓘ **Note**
>
> If you have the same non-key fields in both of your data sources, and the data within those fields does not match, then the index will contain the values from whichever indexer ran most recently. In our example, both data sources contain a **HotelName** field. If for some reason the data in this field is different, for documents with the same key value, then the **HotelName** data from the data source that was indexed most recently will be the value stored in the index.

5 - Search

You can explore the populated search index after the program has run, using the **Search explorer** in the portal.

In Azure portal, open the search service **Overview** page, and find the **hotel-rooms-sample** index in the **Indexes** list.

Select on the hotel-rooms-sample index in the list. You'll see a Search Explorer interface for the index. Enter a query for a term like "Luxury". You should see at least one document in the results, and this document should show a list of room objects in its rooms array.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code checks for existing objects and deletes or updates them so that you can rerun the program.

You can also use the portal to delete indexes, indexers, and data sources.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with the concept of ingesting data from multiple sources, let's take a closer look at indexer configuration, starting with Azure Cosmos DB.

Configure an Azure Cosmos DB indexer

Additional resources

Documentation

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Create Indexer (Azure Cognitive Search REST API-version=2020-06-30)

Indexers are resources that automate many aspects of data ingestion into an Azure Cognitive Search indexes. You must use a supported Azure data source to use this API.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Create an indexer - Azure Cognitive Search

Set properties on an indexer to determine data origin and destinations. You can set parameters to modify runtime behaviors.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

REST samples - Azure Cognitive Search

Find Azure Cognitive Search demo REST code samples that use the Search or Management REST APIs.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Show 5 more

Training

Module

Search data outside the Azure platform in Azure Cognitive Search using Azure Data Factory - Training

Use Azure Data Factory to add data that resides inside or outside the Azure platform into your search indexes.

Certification

Microsoft Certified: Azure Cosmos DB Developer Specialty - Certifications

Azure Cosmos DB developers design and implement data models and data distribution, load data into an Azure Cosmos DB database, and optimize and maintain the solution.

Tutorial: Index and enrich encrypted blobs for full-text search in Azure Cognitive Search

Article • 03/24/2023 • 8 minutes to read

This tutorial shows you how to use Azure Cognitive Search to index documents that have been previously encrypted with a customer-managed key in Azure Blob Storage.

Normally, an indexer can't extract content from encrypted files because it doesn't have access to the customer-managed encryption key in Azure Key Vault. However, by leveraging the DecryptBlobFile custom skill, followed by the Document Extraction skill, you can provide controlled access to the key to decrypt the files and then extract content from them. This unlocks the ability to index and enrich these documents without compromising the encryption status of your stored documents.

Starting with previously encrypted whole documents (unstructured text) such as PDF, HTML, DOCX, and PPTX in Azure Blob Storage, this tutorial uses Postman and the Search REST APIs to perform the following tasks:

- ✓ Define a pipeline that decrypts the documents and extracts text from them.
- ✓ Define an index to store the output.
- ✓ Execute the pipeline to create and load the index.
- ✓ Explore results using full text search and a rich query syntax.

If you don't have an Azure subscription, open a free account before you begin.

Prerequisites

- Azure Cognitive Search on any tier or region.
- Azure Storage, Standard performance (general-purpose v2)
- Blobs encrypted with a customer-managed key. See Tutorial: Encrypt and decrypt blobs using Azure Key Vault if you need to create sample data.
- Azure Key Vault in the same subscription as Azure Cognitive Search. The key vault must have **soft-delete** and **purge protection** enabled.
- Postman app

Custom skill deployment creates an Azure Function app and an Azure Storage account. Since these resources are created for you, they aren't listed as a prerequisite. When you're finished with this tutorial, remember to clean up the resources so that you aren't billed for services you're not using.

> **Note**
>
> Skillsets often require **attaching a Cognitive Services resource**. As written, this skillset has no dependency on Cognitive Services and thus no key is required. If you later add enrichments that invoke built-in skills, remember to update your skillset accordingly.

1 - Create services and collect credentials

Deploy the custom skill

This example uses the sample DecryptBlobFile project from the Azure Search Power Skills GitHub repository. In this section, you will deploy the skill to an Azure Function so that it can be used in a skillset. A built-in deployment script creates an Azure Function resource named starting with **psdbf-function-app-** and loads the skill. You'll be prompted to provide a subscription and resource group. Be sure to choose the same subscription that your Azure Key Vault instance lives in.

Operationally, the DecryptBlobFile skill takes the URL and SAS token for each blob as inputs, and it outputs the downloaded, decrypted file using the file reference contract that Azure Cognitive Search expects. Recall that DecryptBlobFile needs the encryption key to perform the decryption. As part of setup, you'll also create an access policy that grants DecryptBlobFile function access to the encryption key in Azure Key Vault.

1. Click the **Deploy to Azure** button found on the DecryptBlobFile landing page, which will open the provided Resource Manager template within the Azure portal.

2. Choose the same subscription where your Azure Key Vault instance exists (this tutorial will not work if you select a different subscription).

3. Select an existing resource group or create a new one. A dedicated resource group makes cleanup easier later.

4. Select **Review + create**, make sure you agree to the terms, and then select **Create** to deploy the Azure Function.

5. Wait for the deployment to finish.

You should have an Azure Function app that contains the decryption logic and an Azure Storage resource that will store application data. In the next several steps, you'll give the app permissions to access the key vault and collect information that you'll need for the REST calls.

Grant permissions in Azure Key Vault

1. Navigate to your Azure Key Vault service in the portal. Create an access policy in the Azure Key Vault that grants key access to the custom skill.

2. On the left navigation pane, select **Access policies**, and then select **+ Create** to start the **Create an access policy** wizard.

3. On the **Permissions** page under **Configure from template**, select **Azure Data Lake Storage or Azure Storage**.

4. Select **Next**.

5. On the **Principal** page, select the Azure Function instance that you deployed. You can search for it using the resource prefix that was used to create it in step 2, which has a default prefix value of **psdbf-function-app**.

6. Select **Next**.

7. On **Review + create**, select **Create**.

Collect app information

1. Navigate to the **psdbf-function-app** function in the portal, and make a note of the following properties you'll need for the REST calls:

2. Get the function URL, which can be found under **Essentials** on the main page for the function.

3. Get the host key code, which can be found by navigating to **App keys**, clicking to show the **default** key, and copying the value.

Get an admin api-key and URL for Azure Cognitive Search

1. Sign in to the Azure portal, and in your search service **Overview** page, get the name of your search service. You can confirm your service name by reviewing the endpoint URL. If your endpoint URL were `https://mydemo.search.windows.net`, your service name would be `mydemo`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key in the header of every request sent to your service. A valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Set up Postman

Install and set up Postman.

Download and install Postman

1. Download the Postman collection source code.

2. Select **File** > **Import** to import the source code into Postman.

3. Select the **Collections** tab, and then select the **...** (ellipsis) button.

4. Select **Edit**.

5. In the **Edit** dialog box, select the **Variables** tab.

6. On the **Variables** tab, provide the values that you've collected in the previous steps. Postman swaps in a value every time it encounters a specific variable inside double braces. For example, Postman replaces the symbol `{{admin-key}}` with the current value that you set for the search service admin API key.

Variable	Where to get it
`admin-key`	On the **Keys** page of the Azure Cognitive Search service.
`search-service-name`	The name of the Azure Cognitive Search service. The URL is `https://{{search-service-name}}.search.windows.net`.
`storage-connection-string`	In the storage account, on the **Access Keys** tab, select **key1** > **Connection string**.
`storage-container-name`	The name of the blob container that has the encrypted files to be indexed.
`function-uri`	In the Azure Function under **Essentials** on the main page.
`function-code`	In the Azure Function, by navigating to **App keys**, clicking to show the **default** key, and copying the value.
`api-version`	Leave as **2020-06-30**.

Variable	Where to get it
`datasource-name`	Leave as **encrypted-blobs-ds**.
`index-name`	Leave as **encrypted-blobs-idx**.
`skillset-name`	Leave as **encrypted-blobs-ss**.
`indexer-name`	Leave as **encrypted-blobs-ixr**.

Review and run each request

In this section, you'll issue four HTTP requests:

- **PUT request to create the index**: This search index holds the data that Azure Cognitive Search uses and returns.
- **POST request to create the data source**: This data source specifies the connection to your storage account containing the encrypted blob files.
- **PUT request to create the skillset**: The skillset specifies the custom skill definition for the Azure Function that will decrypt the blob file data, and a DocumentExtractionSkill to extract the text from each document after it has been decrypted.
- **PUT request to create the indexer**: Running the indexer retrieves the blobs, applies the skillset, and indexes and stores the results. You must run this request last. The custom skill in the skillset invokes the decryption logic.

To issue the requests, in Postman, select the tab for the requests and select **Send** for each of them.

3 - Monitor indexing

Indexing and enrichment commence as soon as you submit the Create Indexer request. Depending on how many documents are in your storage account, indexing can take a while. To find out whether the indexer is still running, use the **Get Indexer Status** request provided as part of the Postman collection and review the response to learn whether the indexer is running, or to view error and warning information.

If you are using the Free tier, the following message is expected: `"Could not extract content or metadata from your document. Truncated extracted text to '32768'`

`characters"`. This message appears because blob indexing on the Free tier has a 32K limit on character extraction. You won't see this message for this data set on higher tiers.

4 - Search

After indexer execution is finished, you can run some queries to verify that the data has been successfully decrypted and indexed. Navigate to your Azure Cognitive Search service in the portal, and use the search explorer to run queries over the indexed data.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you have successfully indexed encrypted files, you can iterate on this pipeline by adding more cognitive skills. This will allow you to enrich and gain additional insights to your data.

If you are working with doubly encrypted data, you might want to investigate the index encryption features available in Azure Cognitive Search. Although the indexer needs decrypted data for indexing purposes, once the index exists, it can be encrypted in a search index using a customer-managed key. This will ensure that your data is always encrypted when at rest. For more information, see Configure customer-managed keys for data encryption in Azure Cognitive Search.

Tutorial: Optimize indexing with the push API

Article • 01/06/2023 • 13 minutes to read

Azure Cognitive Search supports two basic approaches for importing data into a search index: *pushing* your data into the index programmatically, or pointing an Azure Cognitive Search indexer at a supported data source to *pull* in the data.

This tutorial describes how to efficiently index data using the push model by batching requests and using an exponential backoff retry strategy. You can download and run the sample application. This article explains the key aspects of the application and factors to consider when indexing data.

This tutorial uses C# and the .NET SDK to perform the following tasks:

- ✓ Create an index
- ✓ Test various batch sizes to determine the most efficient size
- ✓ Index batches asynchronously
- ✓ Use multiple threads to increase indexing speeds
- ✓ Use an exponential backoff retry strategy to retry failed documents

If you don't have an Azure subscription, create a free account before you begin.

Prerequisites

The following services and tools are required for this tutorial.

- Visual Studio, any edition. Sample code and instructions were tested on the free Community edition.
- Create an Azure Cognitive Search service or find an existing service under your current subscription.

Download files

Source code for this tutorial is in the optimzize-data-indexing/v11 folder in the Azure-Samples/azure-search-dotnet-samples GitHub repository.

Key considerations

When pushing data into an index, there's several key considerations that impact indexing speeds. You can learn more about these factors in the index large data sets article.

Six key factors to consider are:

- **Service tier and number of partitions/replicas** - Adding partitions and increasing your tier will both increase indexing speeds.
- **Index Schema** - Adding fields and adding additional properties to fields (such as *searchable*, *facetable*, or *filterable*) both reduce indexing speeds.
- **Batch size** - The optimal batch size varies based on your index schema and dataset.
- **Number of threads/workers** - a single thread won't take full advantage of indexing speeds
- **Retry strategy** - An exponential backoff retry strategy should be used to optimize indexing.
- **Network data transfer speeds** - Data transfer speeds can be a limiting factor. Index data from within your Azure environment to increase data transfer speeds.

1 - Create Azure Cognitive Search service

To complete this tutorial, you'll need an Azure Cognitive Search service, which you can create in the portal. We recommend using the same tier you plan to use in production so that you can accurately test and optimize indexing speeds.

Get an admin api-key and URL for Azure Cognitive Search

API calls require the service URL and an access key. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

2 - Set up your environment

1. Start Visual Studio and open **OptimizeDataIndexing.sln**.
2. In Solution Explorer, open **appsettings.json** to provide connection information.

JSON

```
{
  "SearchServiceUri": "https://{service-name}.search.windows.net",
  "SearchServiceAdminApiKey": "",
  "SearchIndexName": "optimize-indexing"
}
```

3 - Explore the code

Once you update *appsettings.json*, the sample program in **OptimizeDataIndexing.sln** should be ready to build and run.

This code is derived from the C# Quickstart. You can find more detailed information on the basics of working with the .NET SDK in that article.

This simple C#/.NET console app performs the following tasks:

- Creates a new index based on the data structure of the C# Hotel class (which also references the Address class).
- Tests various batch sizes to determine the most efficient size
- Indexes data asynchronously

- Using multiple threads to increase indexing speeds
- Using an exponential backoff retry strategy to retry failed items

Before running the program, take a minute to study the code and the index definitions for this sample. The relevant code is in several files:

- **Hotel.cs** and **Address.cs** contains the schema that defines the index
- **DataGenerator.cs** contains a simple class to make it easy to create large amounts of hotel data
- **ExponentialBackoff.cs** contains code to optimize the indexing process as described below
- **Program.cs** contains functions that create and delete the Azure Cognitive Search index, indexes batches of data, and tests different batch sizes

Creating the index

This sample program uses the .NET SDK to define and create an Azure Cognitive Search index. It takes advantage of the `FieldBuilder` class to generate an index structure from a C# data model class.

The data model is defined by the Hotel class, which also contains references to the Address class. The FieldBuilder drills down through multiple class definitions to generate a complex data structure for the index. Metadata tags are used to define the attributes of each field, such as whether it's searchable or sortable.

The following snippets from the **Hotel.cs** file show how a single field, and a reference to another data model class, can be specified.

```C#
. . .
[SearchableField(IsSortable = true)]
public string HotelName { get; set; }
. . .
public Address Address { get; set; }
. . .
```

In the **Program.cs** file, the index is defined with a name and a field collection generated by the `FieldBuilder.Build(typeof(Hotel))` method, and then created as follows:

```C#
private static async Task CreateIndexAsync(string indexName,
SearchIndexClient indexClient)
{
```

```
    // Create a new search index structure that matches the properties of
the Hotel class.
    // The Address class is referenced from the Hotel class. The
FieldBuilder
    // will enumerate these to create a complex data structure for the
index.
    FieldBuilder builder = new FieldBuilder();
    var definition = new SearchIndex(indexName,
builder.Build(typeof(Hotel)));

    await indexClient.CreateIndexAsync(definition);
}
```

Generating data

A simple class is implemented in the **DataGenerator.cs** file to generate data for testing. The sole purpose of this class is to make it easy to generate a large number of documents with a unique ID for indexing.

To get a list of 100,000 hotels with unique IDs, you'd run the following lines of code:

```
C#

long numDocuments = 100000;
DataGenerator dg = new DataGenerator();
List<Hotel> hotels = dg.GetHotels(numDocuments, "large");
```

There are two sizes of hotels available for testing in this sample: **small** and **large**.

The schema of your index can have a significant impact on indexing speeds. Because of this impact, it makes sense to convert this class to generate data matching your intended index schema after you run through this tutorial.

4 - Test batch sizes

Azure Cognitive Search supports the following APIs to load single or multiple documents into an index:

- Add, Update, or Delete Documents (REST API)
- IndexDocumentsAction class or IndexDocumentsBatch class

Indexing documents in batches will significantly improve indexing performance. These batches can be up to 1000 documents, or up to about 16 MB per batch.

Determining the optimal batch size for your data is a key component of optimizing indexing speeds. The two primary factors influencing the optimal batch size are:

- The schema of your index
- The size of your data

Because the optimal batch size is dependent on your index and your data, the best approach is to test different batch sizes to determine what results in the fastest indexing speeds for your scenario.

The following function demonstrates a simple approach to testing batch sizes.

C#

```
public static async Task TestBatchSizesAsync(SearchClient searchClient, int
min = 100, int max = 1000, int step = 100, int numTries = 3)
{
    DataGenerator dg = new DataGenerator();

    Console.WriteLine("Batch Size \t Size in MB \t MB / Doc \t Time (ms) \t
MB / Second");
    for (int numDocs = min; numDocs <= max; numDocs += step)
    {
        List<TimeSpan> durations = new List<TimeSpan>();
        double sizeInMb = 0.0;
        for (int x = 0; x < numTries; x++)
        {
            List<Hotel> hotels = dg.GetHotels(numDocs, "large");

            DateTime startTime = DateTime.Now;
            await UploadDocumentsAsync(searchClient,
hotels).ConfigureAwait(false);
            DateTime endTime = DateTime.Now;
            durations.Add(endTime - startTime);

            sizeInMb = EstimateObjectSize(hotels);
        }

        var avgDuration = durations.Average(timeSpan =>
timeSpan.TotalMilliseconds);
        var avgDurationInSeconds = avgDuration / 1000;
        var mbPerSecond = sizeInMb / avgDurationInSeconds;

        Console.WriteLine("{0} \t\t {1} \t\t {2} \t\t {3} \t {4}", numDocs,
Math.Round(sizeInMb, 3), Math.Round(sizeInMb / numDocs, 3),
Math.Round(avgDuration, 3), Math.Round(mbPerSecond, 3));

        // Pausing 2 seconds to let the search service catch its breath
        Thread.Sleep(2000);
    }
```

```
    Console.WriteLine();
}
```

Because not all documents are the same size (although they are in this sample), we estimate the size of the data we're sending to the search service. We do this using the function below that first converts the object to json and then determines its size in bytes. This technique allows us to determine which batch sizes are most efficient in terms of MB/s indexing speeds.

C#

```
// Returns size of object in MB
public static double EstimateObjectSize(object data)
{
    // converting object to byte[] to determine the size of the data
    BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    byte[] Array;

    // converting data to json for more accurate sizing
    var json = JsonSerializer.Serialize(data);
    bf.Serialize(ms, json);
    Array = ms.ToArray();

    // converting from bytes to megabytes
    double sizeInMb = (double)Array.Length / 1000000;

    return sizeInMb;
}
```

The function requires an `SearchClient` as well as the number of tries you'd like to test for each batch size. As there may be some variability in indexing times for each batch, we try each batch three times by default to make the results more statistically significant.

C#

```
await TestBatchSizesAsync(searchClient, numTries: 3);
```

When you run the function, you should see an output like below in your console:

```
Deleting index...

Creating index...

Finding optimal batch size...

Batch Size       Size in MB      MB / Doc        Time (ms)       MB / Second
100              0.29            0.003           241.517         1.202
200              0.58            0.003           279.182         2.079
300              0.871           0.003           434.859         2.003
400              1.161           0.003           506.927         2.291
500              1.452           0.003           593.79          2.445
600              1.742           0.003           752.854         2.314
700              2.032           0.003           862.523         2.356
800              2.323           0.003           929.534         2.499
900              2.614           0.003           1082.359        2.415
1000             2.904           0.003           1255.456        2.313

Complete.  Press any key to end application...
```

Identify which batch size is most efficient and then use that batch size in the next step of the tutorial. You may see a plateau in MB/s across different batch sizes.

5 - Index data

Now that we've identified the batch size we intend to use, the next step is to begin to index the data. To index data efficiently, this sample:

- Uses multiple threads/workers.
- Implements an exponential backoff retry strategy.

Use multiple threads/workers

To take full advantage of Azure Cognitive Search's indexing speeds, you'll likely need to use multiple threads to send batch indexing requests concurrently to the service.

Several of the key considerations mentioned above impact the optimal number of threads. You can modify this sample and test with different thread counts to determine the optimal thread count for your scenario. However, as long as you have several threads running concurrently, you should be able to take advantage of most of the efficiency gains.

As you ramp up the requests hitting the search service, you may encounter HTTP status codes indicating the request didn't fully succeed. During indexing, two common HTTP status codes are:

- **503 Service Unavailable** - This error means that the system is under heavy load and your request can't be processed at this time.
- **207 Multi-Status** - This error means that some documents succeeded, but at least one failed.

Implement an exponential backoff retry strategy

If a failure happens, requests should be retried using an exponential backoff retry strategy.

Azure Cognitive Search's .NET SDK automatically retries 503s and other failed requests but you'll need to implement your own logic to retry 207s. Open-source tools such as Polly can also be used to implement a retry strategy.

In this sample, we implement our own exponential backoff retry strategy. To implement this strategy, we start by defining some variables including the `maxRetryAttempts` and the initial `delay` for a failed request:

C#

```
// Create batch of documents for indexing
var batch = IndexDocumentsBatch.Upload(hotels);

// Create an object to hold the result
IndexDocumentsResult result = null;

// Define parameters for exponential backoff
int attempts = 0;
TimeSpan delay = delay = TimeSpan.FromSeconds(2);
int maxRetryAttempts = 5;
```

The results of the indexing operation are stored in the variable `IndexDocumentResult result`. This variable is important because it allows you to check if any documents in the batch failed as shown below. If there is a partial failure, a new batch is created based on the failed documents' ID.

`RequestFailedException` exceptions should also be caught as they indicate the request failed completely and should also be retried.

C#

```
// Implement exponential backoff
do
{
    try
    {
```

```
        attempts++;
        result = await
searchClient.IndexDocumentsAsync(batch).ConfigureAwait(false);

        var failedDocuments = result.Results.Where(r => r.Succeeded !=
true).ToList();

        // handle partial failure
        if (failedDocuments.Count > 0)
        {
            if (attempts == maxRetryAttempts)
            {
                Console.WriteLine("[MAX RETRIES HIT] - Giving up on the
batch starting at {0}", id);
                break;
            }
            else
            {
                Console.WriteLine("[Batch starting at doc {0} had partial
failure]", id);
                Console.WriteLine("[Retrying {0} failed documents] \n",
failedDocuments.Count);

                // creating a batch of failed documents to retry
                var failedDocumentKeys = failedDocuments.Select(doc =>
doc.Key).ToList();
                hotels = hotels.Where(h =>
failedDocumentKeys.Contains(h.HotelId)).ToList();
                batch = IndexDocumentsBatch.Upload(hotels);

                Task.Delay(delay).Wait();
                delay = delay * 2;
                continue;
            }
        }

        return result;
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("[Batch starting at doc {0} failed]", id);
        Console.WriteLine("[Retrying entire batch] \n");

        if (attempts == maxRetryAttempts)
        {
            Console.WriteLine("[MAX RETRIES HIT] - Giving up on the batch
starting at {0}", id);
            break;
        }

        Task.Delay(delay).Wait();
        delay = delay * 2;
    }
} while (true);
```

From here, we wrap the exponential backoff code into a function so it can be easily called.

Another function is then created to manage the active threads. For simplicity, that function isn't included here but can be found in ExponentialBackoff.cs. The function can be called with the following command where `hotels` is the data we want to upload, `1000` is the batch size, and `8` is the number of concurrent threads:

```
C#

await ExponentialBackoff.IndexData(indexClient, hotels, 1000, 8);
```

When you run the function, you should see an output like below:

```
Deleting index...

Creating index...

Uploading using exponential backoff...

Uploading 100000 documents...

Creating 8 threads...

Sending a batch of 1000 docs starting with doc 0...

Sending a batch of 1000 docs starting with doc 1000...

Sending a batch of 1000 docs starting with doc 2000...

Sending a batch of 1000 docs starting with doc 3000...

Sending a batch of 1000 docs starting with doc 4000...

Sending a batch of 1000 docs starting with doc 5000...

Sending a batch of 1000 docs starting with doc 6000...

Sending a batch of 1000 docs starting with doc 7000...

Finished a thread, kicking off another...
Sending a batch of 1000 docs starting with doc 8000...
```

When a batch of documents fails, an error is printed out indicating the failure and that the batch is being retried:

```
[Batch starting at doc 6000 had partial failure]
[Retrying 560 failed documents]
```

After the function is finished running, you can verify that all of the documents were added to the index.

6 - Explore index

You can explore the populated search index after the program has run programatically or using the **Search explorer** in the portal.

Programatically

There are two main options for checking the number of documents in an index: the Count Documents API and the Get Index Statistics API. Both paths may require some additional time to update so don't be alarmed if the number of documents returned is lower than you expected initially.

Count Documents

The Count Documents operation retrieves a count of the number of documents in a search index:

```C#
long indexDocCount = await searchClient.GetDocumentCountAsync();
```

Get Index Statistics

The Get Index Statistics operation returns a document count for the current index, plus storage usage. Index statistics will take longer than document count to update.

```C#
var indexStats = await indexClient.GetIndexStatisticsAsync(indexName);
```

Azure portal

In Azure portal, open the search service **Overview** page, and find the **optimize-indexing** index in the **Indexes** list.

Usage Monitoring **Indexes** Indexers Data sources Skillsets

Name	Document Count	Storage Size
optimize-indexing	100,000	12.41 MB

The *Document Count* and *Storage Size* are based on Get Index Statistics API and may take several minutes to update.

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing indexes and deletes them so that you can rerun your code.

You can also use the portal to delete indexes.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

Next steps

Now that you're familiar with the concept of ingesting data efficiently, let's take a closer look at Lucene query architecture and how full text search works in Azure Cognitive Search.

How full text search works in Azure Cognitive Search

Additional resources

Documentation

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Show 5 more

Tutorial: Index large data from Apache Spark using SynapseML and Cognitive Search

Article • 02/01/2023 • 8 minutes to read

In this Azure Cognitive Search tutorial, learn how to index and query large data loaded from a Spark cluster. You'll set up a Jupyter Notebook that performs the following actions:

- ✓ Load various forms (invoices) into a data frame in an Apache Spark session
- ✓ Analyze them to determine their features
- ✓ Assemble the resulting output into a tabular data structure
- ✓ Write the output to a search index hosted in Azure Cognitive Search
- ✓ Explore and query over the content you created

This tutorial takes a dependency on SynapseML, an open source library that supports massively parallel machine learning over big data. In SynapseML, search indexing and machine learning are exposed through *transformers* that perform specialized tasks. Transformers tap into a wide range of AI capabilities. In this exercise, you'll use the **AzureSearchWriter** APIs for analysis and AI enrichment.

Although Azure Cognitive Search has native AI enrichment, this tutorial shows you how to access AI capabilities outside of Cognitive Search. By using SynapseML instead of indexers or skills, you're not subject to data limits or other constraints associated with those objects.

> **Tip**
>
> Watch a short video of this demo at **https://www.youtube.com/watch?v=iXnBLwp7f88**. The video expands on this tutorial with more steps and visuals.

Prerequisites

You'll need the `synapseml` library and several Azure resources. If possible, use the same subscription and region for your Azure resources and put everything into one resource group for simple cleanup later. The following links are for portal installs. The sample data is imported from a public site.

- SynapseML package [1]

- Azure Cognitive Search (any tier) [2]
- Azure Cognitive Services (any tier) [3]
- Azure Databricks (any tier) [4]

[1] This link resolves to a tutorial for loading the package.

[2] You can use the free search tier to index the sample data, but choose a higher tier if your data volumes are large. For non-free tiers, you'll need to provide the search API key in the Set up dependencies step further on.

[3] This tutorial uses Azure Forms Recognizer and Azure Translator. In the instructions that follow, you'll provide a Cognitive Services multi-service key and the region, and it will work for both services.

[4] In this tutorial, Azure Databricks provides the Spark computing platform and the instructions in the link will tell you how to set up the workspace. For this tutorial, we used the portal steps in "Create a workspace".

> **Note**
>
> All of the above Azure resources support security features in the Microsoft Identity platform. For simplicity, this tutorial assumes key-based authentication, using endpoints and keys copied from the portal pages of each service. If you implement this workflow in a production environment, or share the solution with others, remember to replace hard-coded keys with integrated security or encrypted keys.

1 - Create a Spark cluster and notebook

In this section, you'll create a cluster, install the `synapseml` library, and create a notebook to run the code.

1. In Azure portal, find your Azure Databricks workspace and select **Launch workspace**.
2. On the left menu, select **Compute**.
3. Select **Create cluster**.
4. Give the cluster a name, accept the default configuration, and then create the cluster. It takes several minutes to create the cluster.
5. Install the `synapseml` library after the cluster is created:

a. Select **Library** from the tabs at the top of the cluster's page.

b. Select **Install new**.

c. Select **Maven**.

d. In Coordinates, enter `com.microsoft.azure:synapseml_2.12:0.10.0`

e. Select **Install**.

6. On the left menu, select **Create** > **Notebook**.

7. Give the notebook a name, select **Python** as the default language, and select the cluster that has the `synapseml` library.

8. Create seven consecutive cells. You'll paste code into each one.

2 - Set up dependencies

Paste the following code into the first cell of your notebook. Replace the placeholders with endpoints and access keys for each resource. No other modifications are required, so run the code when you're ready.

This code imports multiple packages and sets up access to the Azure resources used in this workflow.

Python

```
import os
from pyspark.sql.functions import udf, trim, split, explode, col,
monotonically_increasing_id, lit
from pyspark.sql.types import StringType
from synapse.ml.core.spark import FluentAPI

cognitive_services_key = "placeholder-cognitive-services-multi-service-key"
```

```
cognitive_services_region = "placeholder-cognitive-services-region"

search_service = "placeholder-search-service-name"
search_key = "placeholder-search-service-api-key"
search_index = "placeholder-search-index-name"
```

3 - Load data into Spark

Paste the following code into the second cell. No modifications are required, so run the code when you're ready.

This code loads a few external files from an Azure storage account that's used for demo purposes. The files are various invoices, and they're read into a data frame.

Python

```python
def blob_to_url(blob):
    [prefix, postfix] = blob.split("@")
    container = prefix.split("/")[-1]
    split_postfix = postfix.split("/")
    account = split_postfix[0]
    filepath = "/".join(split_postfix[1:])
    return "https://{}/{}/{}".format(account, container, filepath)


df2 = (spark.read.format("binaryFile")

.load("wasbs://ignite2021@mmlsparkdemo.blob.core.windows.net/form_subset/*")
    .select("path")
    .limit(10)
    .select(udf(blob_to_url, StringType())("path").alias("url"))
    .cache())

display(df2)
```

4 - Add form recognition

Paste the following code into the third cell. No modifications are required, so run the code when you're ready.

This code loads the AnalyzeInvoices transformer and passes a reference to the data frame containing the invoices. It calls the pre-built invoice model of Azure Forms Analyzer.

Python

```
from synapse.ml.cognitive import AnalyzeInvoices

analyzed_df = (AnalyzeInvoices()
    .setSubscriptionKey(cognitive_services_key)
    .setLocation(cognitive_services_region)
    .setImageUrlCol("url")
    .setOutputCol("invoices")
    .setErrorCol("errors")
    .setConcurrency(5)
    .transform(df2)
    .cache())

display(analyzed_df)
```

The output from this step should look similar to the next screenshot. Notice how the forms analysis is packed into a densely structured column, which is difficult to work with. The next transformation resolves this issue by parsing the column into rows and columns.

5 - Restructure form recognition output

Paste the following code into the fourth cell and run it. No modifications are required.

This code loads FormOntologyLearner, a transformer that analyzes the output of Form Recognizer transformers and infers a tabular data structure. The output of AnalyzeInvoices is dynamic and varies based on the features detected in your content. Furthermore, the transformer consolidates output into a single column. Because the output is dynamic and consolidated, it's difficult to use in downstream transformations that require more structure.

FormOntologyLearner extends the utility of the AnalyzeInvoices transformer by looking for patterns that can be used to create a tabular data structure. Organizing the output

into multiple columns and rows makes the content consumable in other transformers, like AzureSearchWriter.

Python

```
from synapse.ml.cognitive import FormOntologyLearner

itemized_df = (FormOntologyLearner()
    .setInputCol("invoices")
    .setOutputCol("extracted")
    .fit(analyzed_df)
    .transform(analyzed_df)
    .select("url", "extracted.*").select("*",
explode(col("Items")).alias("Item"))
    .drop("Items").select("Item.*", "*").drop("Item"))

display(itemized_df)
```

Notice how this transformation recasts the nested fields into a table, which enables the next two transformations. This screenshot is trimmed for brevity. If you're following along in your own notebook, you'll have 19 columns and 26 rows.

Table Data Profile

	ProductCode	Tax	Quantity	UnitPrice	Description	Amount	url
1	61	3.78	1	54	One sat on shoe gnome	57.78	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
2	23	17.22	2	123	White Door	263.22	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
3	42	5.67	1	90	Metal Shelving	86.67	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117876.pdf
4	35	4.2	2	30	Wood Pack	64.2	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
5	45	30.45	3	145	Wooden Saw	465.45	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
6	23	25.83	3	123	White Door	394.83	https://mmlsparkdemo.blob.core.windows.net/ignite2021/form_subset/Invoice117874.pdf
7							

Showing all 26 rows.

6 - Add translations

Paste the following code into the fifth cell. No modifications are required, so run the code when you're ready.

This code loads Translate, a transformer that calls the Azure Translator service in Cognitive Services. The original text, which is in English in the "Description" column, is machine-translated into various languages. All of the output is consolidated into "output.translations" array.

Python

```
from synapse.ml.cognitive import Translate

translated_df = (Translate()
    .setSubscriptionKey(cognitive_services_key)
    .setLocation(cognitive_services_region)
    .setTextCol("Description")
```

```
    .setErrorCol("TranslationError")
    .setOutputCol("output")
    .setToLanguage(["zh-Hans", "fr", "ru", "cy"])
    .setConcurrency(5)
    .transform(itemized_df)
    .withColumn("Translations", col("output.translations")[0])
    .drop("output", "TranslationError")
    .cache())

display(translated_df)
```

Tip

To check for translated strings, scroll to the end of the rows.

7 - Add a search index with AzureSearchWriter

Paste the following code in the sixth cell and then run it. No modifications are required.

This code loads AzureSearchWriter. It consumes a tabular dataset and infers a search index schema that defines one field for each column. The translations structure is an array, so it's articulated in the index as a complex collection with subfields for each language translation. The generated index will have a document key and use the default values for fields created using the Create Index REST API.

Python

```
from synapse.ml.cognitive import *

(translated_df.withColumn("DocID",
monotonically_increasing_id().cast("string"))
    .withColumn("SearchAction", lit("upload"))
    .writeToAzureSearch(
        subscriptionKey=search_key,
        actionCol="SearchAction",
        serviceName=search_service,
        indexName=search_index,
        keyCol="DocID",
    ))
```

You can check the search service pages in Azure portal to explore the index definition created by AzureSearchWriter.

> **Note**
>
> If you can't use default search index, you can provide an external custom definition in JSON, passing its URI as a string in the "indexJson" property. Generate the default index first so that you know which fields to specify, and then follow with customized properties if you need specific analyzers, for example.

8 - Query the index

Paste the following code into the seventh cell and then run it. No modifications are required, except that you might want to vary the syntax or try more examples to further explore your content:

- Query syntax
- Query examples

There's no transformer or module that issues queries. This cell is a simple call to the Search Documents REST API.

This particular example is searching for the word "door" (`"search": "door"`). It also returns a "count" of the number of matching documents, and selects just the contents of the "Description' and "Translations" fields for the results. If you want to see the full list of fields, remove the "select" parameter.

Python

```python
import requests

url = "https://{}.search.windows.net/indexes/{}/docs/search?api-version=2020-06-30".format(search_service, search_index)
requests.post(url, json={"search": "door", "count": "true", "select": "Description, Translations"}, headers={"api-key": search_key}).json()
```

The following screenshot shows the cell output for sample script.

```
Cmd 7
import requests

url = "https://{}.search.windows.net/indexes/{}/docs/search?api-version=2020-06-30".format(search_service, search_index)
requests.post(url, json={"search": "door", "count": "true", "select": "Description, Translations"}, headers={"api-key": search_key}).json()
```

```
Out[34]: {'@odata.context': "https://srch-synapse-demo.search.windows.net/indexes('my-index')/$metadata#docs(*)",
 '@odata.count': 2,
 'value': [{'@search.score': 1.4385834,
   'Description': 'White Door',
   'Translations': [{'to': 'zh-Hans',
     'text': '白色门',
     'transliteration': None,
     'alignment': None,
     'sentLen': None},
    {'to': 'fr',
     'text': 'Porte blanche',
     'transliteration': None,
     'alignment': None,
     'sentLen': None},
    {'to': 'ru',
     'text': 'Белая дверь',
     'transliteration': None,
     'alignment': None,
     'sentLen': None},
    {'to': 'cy',
     'text': 'Drws Gwyn',
```

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

Next steps

In this tutorial, you learned about the AzureSearchWriter transformer in SynapseML, which is a new way of creating and loading search indexes in Azure Cognitive Search. The transformer takes structured JSON as an input. The FormOntologyLearner can provide the necessary structure for output produced by the Forms Recognizer transformers in SynapseML.

As a next step, review the other SynapseML tutorials that produce transformed content you might want to explore through Azure Cognitive Search:

Tutorial: Text Analytics with Cognitive Services

Additional resources

Documentation

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Entity Linking cognitive skill (v3) - Azure Cognitive Search

Extract different linked entities from text in an enrichment pipeline in Azure Cognitive Search.

Custom Entity Lookup cognitive search skill - Azure Cognitive Search

Extract different custom entities from text in an Azure Cognitive Search cognitive search pipeline.

Azure Cognitive Search client library for Python

Quickstart: Create a search index in Python - Azure Cognitive Search

Learn how to create a search index, load data, and run queries using Python, Jupyter Notebook, and the Azure.Documents.Search client library for Python.

Show 5 more

Tutorial: Use .NET and AI to generate searchable content from Azure blobs

Article • 02/08/2022 • 22 minutes to read

If you have unstructured text or images in Azure Blob Storage, an AI enrichment pipeline can extract information and create new content for full-text search or knowledge mining scenarios.

In this C# tutorial, you will learn how to:

- ✓ Set up a development environment.
- ✓ Define a pipeline that uses OCR, language detection, entity recognition, and key phrase extraction.
- ✓ Execute the pipeline to invoke transformations, and to create and load a search index.
- ✓ Explore results using full text search and a rich query syntax.

If you don't have an Azure subscription, open a free account before you begin.

Overview

This tutorial uses C# and the **Azure.Search.Documents** client library to create a data source, index, indexer, and skillset.

The indexer connects to sample data in a blob container that's specified in the data source object, and sends all enriched content to a search index.

The skillset is attached to the indexer. It uses built-in skills from Microsoft to find and extract information. Steps in the pipeline include Optical Character Recognition (OCR) on images, language detection on text, key phrase extraction, and entity recognition (organizations). New information created by the pipeline is stored in new fields in an index. Once the index is populated, you can use the fields in queries, facets, and filters.

Prerequisites

- Visual Studio
- Azure.Search.Documents 11.x NuGet package
- Azure Storage
- Azure Cognitive Search
- Sample data

> **Note**
>
> You can use the free search service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

Download files

The sample data consists of 14 files of mixed content type that you will upload to Azure Blob Storage in a later step.

1. Get the files from azure-search-sample-data/ai-enrichment-mixed-media/ and copy them to your local computer.

2. Next, get the source code for this tutorial. Source code is in the **tutorial-ai-enrichment/v11** folder in the azure-search-dotnet-samples repository.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, Cognitive Services on the backend for AI enrichment, and Azure Blob Storage to provide the data. This tutorial stays under the free allocation of 20 transactions per indexer per day on Cognitive Services, so the only services you need to create are search and storage.

If possible, create both in the same region and resource group for proximity and manageability. In practice, your Azure Storage account can be in any region.

Start with Azure Storage

1. Sign in to the Azure portal and click + **Create Resource**.

2. Search for *storage account* and select Microsoft's Storage Account offering.

3. In the Basics tab, the following items are required. Accept the defaults for everything else.

- **Resource group**. Select an existing one or create a new one, but use the same group for all services so that you can manage them collectively.
- **Storage account name**. If you think you might have multiple resources of the same type, use the name to disambiguate by type and region, for example *blobstoragewestus*.
- **Location**. If possible, choose the same location used for Azure Cognitive Search and Cognitive Services. A single location voids bandwidth charges.
- **Account Kind**. Choose the default, *StorageV2 (general purpose v2)*.

4. Select **Review + Create** to create the service.
5. Once it's created, select **Go to the resource** to open the Overview page.
6. Select **Blobs** service.
7. Select **+ Container** to create a container and name it *cog-search-demo*.
8. Select *cog-search-demo* and then select **Upload** to open the folder where you saved the download files. Select all of the files. Select **Upload**.

Name	Date modified	Type	Size
facts-about-microsoft.html	12/8/2021 12:16 PM	Chrome HTML Document	138 KB
guthrie.jpg	12/23/2019 8:52 AM	JPG File	46 KB
redshirt.jpg	12/23/2019 8:52 AM	JPG File	67 KB
satyanadellalinux.jpg	12/23/2019 8:52 AM	JPG File	108 KB
Cognitive Services and Bots (spanish).pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	3,884 KB
increase_green_energy_ one_rooftop_at_a_time.pdf	12/8/2021 12:35 PM	Microsoft Edge PDF Document	499 KB
MSFT_cloud_architecture_contoso.pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	1,805 KB
Cognitive Searvices and Content Intelligence.pptx	12/23/2019 8:52 AM	Microsoft PowerPoint Presenta...	11,231 KB
Mesh_for_Microsoft_Teams.docx	12/8/2021 12:23 PM	Microsoft Word Document	4,137 KB
5074.clip_image002_6FE27E85.png	12/23/2019 8:52 AM	PNG File	472 KB
create-search-collect-info.png	12/23/2019 8:52 AM	PNG File	57 KB
create-search-service.png	12/23/2019 8:52 AM	PNG File	26 KB
create-service-full-portal.png	12/23/2019 8:52 AM	PNG File	67 KB
satyasletter.txt	12/23/2019 8:52 AM	Text Document	6 KB

9. Before you leave Azure Storage, get a connection string so that you can formulate a connection in Azure Cognitive Search.

   a. Browse back to the Overview page of your storage account (we used *blobstragewestus* as an example).

   b. In the left navigation pane, select **Access keys** and copy one of the connection strings.

   The connection string is a URL similar to the following example:

```HTTP
DefaultEndpointsProtocol=https;AccountName=cogsrchdemostorage;AccountKey=<your account key>;EndpointSuffix=core.windows.net
```

10. Save the connection string to Notepad. You'll need it later when setting up the data source connection.

Cognitive Services

AI enrichment is backed by Cognitive Services, including Language service and Computer Vision for natural language and image processing. If your objective was to complete an actual prototype or project, you would at this point provision Cognitive Services (in the same region as Azure Cognitive Search) so that you can attach it to indexing operations.

For this exercise, however, you can skip resource provisioning because Azure Cognitive Search can connect to Cognitive Services behind the scenes and give you 20 free transactions per indexer run. Since this tutorial uses 14 transactions, the free allocation is sufficient. For larger projects, plan on provisioning Cognitive Services at the pay-as-you-go S0 tier. For more information, see Attach Cognitive Services.

Azure Cognitive Search

The third component is Azure Cognitive Search, which you can create in the portal or find an existing search service in your subscription.

You can use the Free tier to complete this walkthrough.

Copy an admin api-key and URL for Azure Cognitive Search

To interact with your Azure Cognitive Search service you will need the service URL and an access key.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the name of your search service. You can confirm your service name by reviewing the endpoint URL. If your endpoint URL were `https://mydemo.search.windows.net`, your service name would be `mydemo`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. You can copy either the primary or secondary key.

Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Set up your environment

Begin by opening Visual Studio and creating a new Console App project that can run on .NET Core.

Install Azure.Search.Documents

The Azure Cognitive Search .NET SDK consists of a client library that enables you to manage your indexes, data sources, indexers, and skillsets, as well as upload and manage documents and execute queries, all without having to deal with the details of HTTP and JSON. This client library is distributed as a NuGet package.

For this project, install version 11 or later of the `Azure.Search.Documents` and the latest version of `Microsoft.Extensions.Configuration`.

1. In Visual Studio, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution...**
2. Browse for Azure.Search.Document.
3. Select the latest version and then click **Install**.

4. Repeat the previous steps to install Microsoft.Extensions.Configuration and Microsoft.Extensions.Configuration.Json.

Add service connection information

1. Right-click on your project in the Solution Explorer and select **Add** > **New Item...** .

2. Name the file `appsettings.json` and select **Add**.

3. Include this file in your output directory.
   a. Right-click on `appsettings.json` and select **Properties**.
   b. Change the value of **Copy to Output Directory** to **Copy if newer**.

4. Copy the below JSON into your new JSON file.

```JSON
{
  "SearchServiceUri": "Put your search service URI here",
  "SearchServiceAdminApiKey": "Put your primary or secondary API key here",
  "SearchServiceQueryApiKey": "Put your query API key here",
  "AzureBlobConnectionString": "Put your Azure Blob connection string here",
}
```

Add your search service and blob storage account information. Recall that you can get this information from the service provisioning steps indicated in the previous section.

For **SearchServiceUri**, enter the full URL.

Add namespaces

In `Program.cs`, add the following namespaces.

```C#
using Azure;
using Azure.Search.Documents.Indexes;
using Azure.Search.Documents.Indexes.Models;
using Microsoft.Extensions.Configuration;
using System;
using System.Collections.Generic;
using System.Linq;

namespace EnrichwithAI
```

Create a client

Create an instance of a `SearchIndexClient` and a `SearchIndexerClient` under `Main`.

C#

```
public static void Main(string[] args)
{
    // Create service client
    IConfigurationBuilder builder = new
ConfigurationBuilder().AddJsonFile("appsettings.json");
    IConfigurationRoot configuration = builder.Build();

    string searchServiceUri = configuration["SearchServiceUri"];
    string adminApiKey = configuration["SearchServiceAdminApiKey"];
    string cognitiveServicesKey = configuration["CognitiveServicesKey"];

    SearchIndexClient indexClient = new SearchIndexClient(new
Uri(searchServiceUri), new AzureKeyCredential(adminApiKey));
    SearchIndexerClient indexerClient = new SearchIndexerClient(new
Uri(searchServiceUri), new AzureKeyCredential(adminApiKey));
}
```

> **Note**
>
> The clients connect to your search service. In order to avoid opening too many connections, you should try to share a single instance in your application if possible. The methods are thread-safe to enable such sharing.

Add function to exit the program during failure

This tutorial is meant to help you understand each step of the indexing pipeline. If there is a critical issue that prevents the program from creating the data source, skillset, index, or indexer the program will output the error message and exit so that the issue can be understood and addressed.

Add `ExitProgram` to `Main` to handle scenarios that require the program to exit.

C#

```
private static void ExitProgram(string message)
{
    Console.WriteLine("{0}", message);
    Console.WriteLine("Press any key to exit the program...");
    Console.ReadKey();
```

```
    Environment.Exit(0);
}
```

3 - Create the pipeline

In Azure Cognitive Search, AI processing occurs during indexing (or data ingestion). This part of the walkthrough creates four objects: data source, index definition, skillset, indexer.

Step 1: Create a data source

`SearchIndexerClient` has a DataSourceName property that you can set to a `SearchIndexerDataSourceConnection` object. This object provides all the methods you need to create, list, update, or delete Azure Cognitive Search data sources.

Create a new `SearchIndexerDataSourceConnection` instance by calling `indexerClient.CreateOrUpdateDataSourceConnection(dataSource)`. The following code creates a data source of type `AzureBlob`.

C#

```
private static SearchIndexerDataSourceConnection
CreateOrUpdateDataSource(SearchIndexerClient indexerClient,
IConfigurationRoot configuration)
{
    SearchIndexerDataSourceConnection dataSource = new
SearchIndexerDataSourceConnection(
        name: "demodata",
        type: SearchIndexerDataSourceType.AzureBlob,
        connectionString: configuration["AzureBlobConnectionString"],
        container: new SearchIndexerDataContainer("cog-search-demo"))
    {
        Description = "Demo files to demonstrate cognitive search
capabilities."
    };

    // The data source does not need to be deleted if it was already created
    // since we are using the CreateOrUpdate method
    try
    {
        indexerClient.CreateOrUpdateDataSourceConnection(dataSource);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Failed to create or update the data source\n
Exception message: {0}\n", ex.Message);
        ExitProgram("Cannot continue without a data source");
```

```
    }

    return dataSource;
}
```

For a successful request, the method will return the data source that was created. If there is a problem with the request, such as an invalid parameter, the method will throw an exception.

Now add a line in `Main` to call the `CreateOrUpdateDataSource` function that you've just added.

C#

```
// Create or Update the data source
Console.WriteLine("Creating or updating the data source...");
SearchIndexerDataSourceConnection dataSource =
CreateOrUpdateDataSource(indexerClient, configuration);
```

Build and run the solution. Since this is your first request, check the Azure portal to confirm the data source was created in Azure Cognitive Search. On the search service overview page, verify the Data Sources list has a new item. You might need to wait a few minutes for the portal page to refresh.

Step 2: Create a skillset

In this section, you define a set of enrichment steps that you want to apply to your data. Each enrichment step is called a *skill* and the set of enrichment steps, a *skillset*. This tutorial uses built-in cognitive skills for the skillset:

- Optical Character Recognition to recognize printed and handwritten text in image files.
- Text Merger to consolidate text from a collection of fields into a single "merged content" field.

- Language Detection to identify the content's language.
- Entity Recognition for extracting the names of organizations from content in the blob container.
- Text Split to break large content into smaller chunks before calling the key phrase extraction skill and the entity recognition skill. Key phrase extraction and entity recognition accept inputs of 50,000 characters or less. A few of the sample files need splitting up to fit within this limit.
- Key Phrase Extraction to pull out the top key phrases.

During initial processing, Azure Cognitive Search cracks each document to extract content from different file formats. Text originating in the source file is placed into a generated `content` field, one for each document. As such, set the input as `"/document/content"` to use this text. Image content is placed into a generated `normalized_images` field, specified in a skillset as `/document/normalized_images/*`.

Outputs can be mapped to an index, used as input to a downstream skill, or both as is the case with language code. In the index, a language code is useful for filtering. As an input, language code is used by text analysis skills to inform the linguistic rules around word breaking.

For more information about skillset fundamentals, see How to define a skillset.

OCR skill

The OcrSkill extracts text from images. This skill assumes that a normalized_images field exists. To generate this field, later in the tutorial we'll set the `"imageAction"` configuration in the indexer definition to `"generateNormalizedImages"`.

C#

```
private static OcrSkill CreateOcrSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("image")
    {
        Source = "/document/normalized_images/*"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("text")
    {
```

```
        TargetName = "text"
    });

    OcrSkill ocrSkill = new OcrSkill(inputMappings, outputMappings)
    {
        Description = "Extract text (plain and structured) from image",
        Context = "/document/normalized_images/*",
        DefaultLanguageCode = OcrSkillLanguage.En,
        ShouldDetectOrientation = true
    };

    return ocrSkill;
}
```

Merge skill

In this section, you'll create a MergeSkill that merges the document content field with the text that was produced by the OCR skill.

C#

```
private static MergeSkill CreateMergeSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/content"
    });
    inputMappings.Add(new InputFieldMappingEntry("itemsToInsert")
    {
        Source = "/document/normalized_images/*/text"
    });
    inputMappings.Add(new InputFieldMappingEntry("offsets")
    {
        Source = "/document/normalized_images/*/contentOffset"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("mergedText")
    {
        TargetName = "merged_text"
    });

    MergeSkill mergeSkill = new MergeSkill(inputMappings, outputMappings)
    {
        Description = "Create merged_text which includes all the textual
representation of each image inserted at the right location in the content
field.",
        Context = "/document",
        InsertPreTag = " ",
```

```
        InsertPostTag = " "
    };

    return mergeSkill;
}
```

Language detection skill

The LanguageDetectionSkill detects the language of the input text and reports a single language code for every document submitted on the request. We'll use the output of the **Language Detection** skill as part of the input to the **Text Split** skill.

C#

```
private static LanguageDetectionSkill CreateLanguageDetectionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/merged_text"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("languageCode")
    {
        TargetName = "languageCode"
    });

    LanguageDetectionSkill languageDetectionSkill = new
LanguageDetectionSkill(inputMappings, outputMappings)
    {
        Description = "Detect the language used in the document",
        Context = "/document"
    };

    return languageDetectionSkill;
}
```

Text split skill

The below SplitSkill will split text by pages and limit the page length to 4,000 characters as measured by `String.Length`. The algorithm will try to split the text into chunks that are at most `maximumPageLength` in size. In this case, the algorithm will do its best to break the sentence on a sentence boundary, so the size of the chunk may be slightly less than `maximumPageLength`.

C#

```
private static SplitSkill CreateSplitSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/merged_text"
    });
    inputMappings.Add(new InputFieldMappingEntry("languageCode")
    {
        Source = "/document/languageCode"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("textItems")
    {
        TargetName = "pages",
    });

    SplitSkill splitSkill = new SplitSkill(inputMappings, outputMappings)
    {
        Description = "Split content into pages",
        Context = "/document",
        TextSplitMode = TextSplitMode.Pages,
        MaximumPageLength = 4000,
        DefaultLanguageCode = SplitSkillLanguage.En
    };

    return splitSkill;
}
```

Entity recognition skill

This `EntityRecognitionSkill` instance is set to recognize category type `organization`. The EntityRecognitionSkill can also recognize category types `person` and `location`.

Notice that the "context" field is set to `"/document/pages/*"` with an asterisk, meaning the enrichment step is called for each page under `"/document/pages"`.

C#

```
private static EntityRecognitionSkill CreateEntityRecognitionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
```

```
        Source = "/document/pages/*"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("organizations")
    {
        TargetName = "organizations"
    });

    EntityRecognitionSkill entityRecognitionSkill = new
EntityRecognitionSkill(inputMappings, outputMappings)
    {
        Description = "Recognize organizations",
        Context = "/document/pages/*",
        DefaultLanguageCode = EntityRecognitionSkillLanguage.En
    };
    entityRecognitionSkill.Categories.Add(EntityCategory.Organization);

    return entityRecognitionSkill;
}
```

Key phrase extraction skill

Like the `EntityRecognitionSkill` instance that was just created, the KeyPhraseExtractionSkill is called for each page of the document.

C#

```
private static KeyPhraseExtractionSkill CreateKeyPhraseExtractionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new
List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry("text")
    {
        Source = "/document/pages/*"
    });
    inputMappings.Add(new InputFieldMappingEntry("languageCode")
    {
        Source = "/document/languageCode"
    });

    List<OutputFieldMappingEntry> outputMappings = new
List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry("keyPhrases")
    {
        TargetName = "keyPhrases"
    });

    KeyPhraseExtractionSkill keyPhraseExtractionSkill = new
KeyPhraseExtractionSkill(inputMappings, outputMappings)
```

```
    {
        Description = "Extract the key phrases",
        Context = "/document/pages/*",
        DefaultLanguageCode = KeyPhraseExtractionSkillLanguage.En
    };

    return keyPhraseExtractionSkill;
}
```

Build and create the skillset

Build the SearchIndexerSkillset using the skills you created.

```C#
private static SearchIndexerSkillset
CreateOrUpdateDemoSkillSet(SearchIndexerClient indexerClient,
IList<SearchIndexerSkill> skills,string cognitiveServicesKey)
{
    SearchIndexerSkillset skillset = new
SearchIndexerSkillset("demoskillset", skills)
    {
        Description = "Demo skillset",
        CognitiveServicesAccount = new
CognitiveServicesAccountKey(cognitiveServicesKey)
    };

    // Create the skillset in your search service.
    // The skillset does not need to be deleted if it was already created
    // since we are using the CreateOrUpdate method
    try
    {
        indexerClient.CreateOrUpdateSkillset(skillset);
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to create the skillset\n Exception
message: {0}\n", ex.Message);
        ExitProgram("Cannot continue without a skillset");
    }

    return skillset;
}
```

Add the following lines to `Main`.

```C#
// Create the skills
Console.WriteLine("Creating the skills...");
```

```
OcrSkill ocrSkill = CreateOcrSkill();
MergeSkill mergeSkill = CreateMergeSkill();
EntityRecognitionSkill entityRecognitionSkill =
CreateEntityRecognitionSkill();
LanguageDetectionSkill languageDetectionSkill =
CreateLanguageDetectionSkill();
SplitSkill splitSkill = CreateSplitSkill();
KeyPhraseExtractionSkill keyPhraseExtractionSkill =
CreateKeyPhraseExtractionSkill();

// Create the skillset
Console.WriteLine("Creating or updating the skillset...");
List<SearchIndexerSkill> skills = new List<SearchIndexerSkill>();
skills.Add(ocrSkill);
skills.Add(mergeSkill);
skills.Add(languageDetectionSkill);
skills.Add(splitSkill);
skills.Add(entityRecognitionSkill);
skills.Add(keyPhraseExtractionSkill);

SearchIndexerSkillset skillset = CreateOrUpdateDemoSkillSet(indexerClient,
skills, cognitiveServicesKey);
```

Step 3: Create an index

In this section, you define the index schema by specifying which fields to include in the searchable index, and the search attributes for each field. Fields have a type and can take attributes that determine how the field is used (searchable, sortable, and so forth). Field names in an index are not required to identically match the field names in the source. In a later step, you add field mappings in an indexer to connect source-destination fields. For this step, define the index using field naming conventions pertinent to your search application.

This exercise uses the following fields and field types:

Field names	Field types
`id`	Edm.String
`content`	Edm.String
`languageCode`	Edm.String
`keyPhrases`	List<Edm.String>
`organizations`	List<Edm.String>

Create DemoIndex Class

The fields for this index are defined using a model class. Each property of the model class has attributes which determine the search-related behaviors of the corresponding index field.

We'll add the model class to a new C# file. Right click on your project and select **Add** > **New Item...**, select "Class" and name the file `DemoIndex.cs`, then select **Add**.

Make sure to indicate that you want to use types from the `Azure.Search.Documents.Indexes` and `System.Text.Json.Serialization` namespaces.

Add the below model class definition to `DemoIndex.cs` and include it in the same namespace where you'll create the index.

C#

```
using Azure.Search.Documents.Indexes;
using System.Text.Json.Serialization;

namespace EnrichwithAI
{
    // The SerializePropertyNamesAsCamelCase is currently unsupported as of
this writing.
    // Replace it with JsonPropertyName
    public class DemoIndex
    {
        [SearchableField(IsSortable = true, IsKey = true)]
        [JsonPropertyName("id")]
        public string Id { get; set; }

        [SearchableField]
        [JsonPropertyName("content")]
        public string Content { get; set; }

        [SearchableField]
        [JsonPropertyName("languageCode")]
        public string LanguageCode { get; set; }

        [SearchableField]
        [JsonPropertyName("keyPhrases")]
        public string[] KeyPhrases { get; set; }

        [SearchableField]
        [JsonPropertyName("organizations")]
        public string[] Organizations { get; set; }
    }
}
```

Now that you've defined a model class, back in `Program.cs` you can create an index definition fairly easily. The name for this index will be `demoindex`. If an index already

exists with that name, it will be deleted.

```C#
private static SearchIndex CreateDemoIndex(SearchIndexClient indexClient)
{
    FieldBuilder builder = new FieldBuilder();
    var index = new SearchIndex("demoindex")
    {
        Fields = builder.Build(typeof(DemoIndex))
    };

    try
    {
        indexClient.GetIndex(index.Name);
        indexClient.DeleteIndex(index.Name);
    }
    catch (RequestFailedException ex) when (ex.Status == 404)
    {
        //if the specified index not exist, 404 will be thrown.
    }

    try
    {
        indexClient.CreateIndex(index);
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to create the index\n Exception message:
{0}\n", ex.Message);
        ExitProgram("Cannot continue without an index");
    }

    return index;
}
```

During testing, you may find that you're attempting to create the index more than once. Because of this, check to see if the index that you're about to create already exists before attempting to create it.

Add the following lines to `Main`.

```C#
// Create the index
Console.WriteLine("Creating the index...");
SearchIndex demoIndex = CreateDemoIndex(indexClient);
```

Add the following using statement to resolve the disambiguate reference.

```C#
using Index = Azure.Search.Documents.Indexes.Models;
```

To learn more about index concepts, see Create Index (REST API).

Step 4: Create and run an indexer

So far you have created a data source, a skillset, and an index. These three components become part of an indexer that pulls each piece together into a single multi-phased operation. To tie these together in an indexer, you must define field mappings.

- The fieldMappings are processed before the skillset, mapping source fields from the data source to target fields in an index. If field names and types are the same at both ends, no mapping is required.
- The outputFieldMappings are processed after the skillset, referencing sourceFieldNames that don't exist until document cracking or enrichment creates them. The targetFieldName is a field in an index.

In addition to hooking up inputs to outputs, you can also use field mappings to flatten data structures. For more information, see How to map enriched fields to a searchable index.

```C#
private static SearchIndexer CreateDemoIndexer(SearchIndexerClient
indexerClient, SearchIndexerDataSourceConnection dataSource,
SearchIndexerSkillset skillSet, SearchIndex index)
{
    IndexingParameters indexingParameters = new IndexingParameters()
    {
        MaxFailedItems = -1,
        MaxFailedItemsPerBatch = -1,
    };
    indexingParameters.Configuration.Add("dataToExtract",
"contentAndMetadata");
    indexingParameters.Configuration.Add("imageAction",
"generateNormalizedImages");

    SearchIndexer indexer = new SearchIndexer("demoindexer",
dataSource.Name, index.Name)
    {
        Description = "Demo Indexer",
        SkillsetName = skillSet.Name,
        Parameters = indexingParameters
    };
```

```
    FieldMappingFunction mappingFunction = new
FieldMappingFunction("base64Encode");
    mappingFunction.Parameters.Add("useHttpServerUtilityUrlTokenEncode",
true);

    indexer.FieldMappings.Add(new FieldMapping("metadata_storage_path")
    {
        TargetFieldName = "id",
        MappingFunction = mappingFunction

    });
    indexer.FieldMappings.Add(new FieldMapping("content")
    {
        TargetFieldName = "content"
    });

    indexer.OutputFieldMappings.Add(new
FieldMapping("/document/pages/*/organizations/*")
    {
        TargetFieldName = "organizations"
    });
    indexer.OutputFieldMappings.Add(new
FieldMapping("/document/pages/*/keyPhrases/*")
    {
        TargetFieldName = "keyPhrases"
    });
    indexer.OutputFieldMappings.Add(new
FieldMapping("/document/languageCode")
    {
        TargetFieldName = "languageCode"
    });

    try
    {
        indexerClient.GetIndexer(indexer.Name);
        indexerClient.DeleteIndexer(indexer.Name);
    }
    catch (RequestFailedException ex) when (ex.Status == 404)
    {
        //if the specified indexer not exist, 404 will be thrown.
    }

    try
    {
        indexerClient.CreateIndexer(indexer);
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to create the indexer\n Exception message:
{0}\n", ex.Message);
        ExitProgram("Cannot continue without creating an indexer");
    }

    return indexer;
}
```

Add the following lines to `Main`.

```C#
// Create the indexer, map fields, and execute transformations
Console.WriteLine("Creating the indexer and executing the pipeline...");
SearchIndexer demoIndexer = CreateDemoIndexer(indexerClient, dataSource,
skillset, demoIndex);
```

Expect indexer processing to take some time to complete. Even though the data set is small, analytical skills are computation-intensive. Some skills, such as image analysis, are long-running.

> **Tip**
>
> Creating an indexer invokes the pipeline. If there are problems reaching the data, mapping inputs and outputs, or order of operations, they appear at this stage.

Explore creating the indexer

The code sets `"maxFailedItems"` to -1, which instructs the indexing engine to ignore errors during data import. This is useful because there are so few documents in the demo data source. For a larger data source, you would set the value to greater than 0.

Also notice the `"dataToExtract"` is set to `"contentAndMetadata"`. This statement tells the indexer to automatically extract the content from different file formats as well as metadata related to each file.

When content is extracted, you can set `imageAction` to extract text from images found in the data source. The `"imageAction"` set to `"generateNormalizedImages"` configuration, combined with the OCR Skill and Text Merge Skill, tells the indexer to extract text from the images (for example, the word "stop" from a traffic Stop sign), and embed it as part of the content field. This behavior applies to both the images embedded in the documents (think of an image inside a PDF), as well as images found in the data source, for instance a JPG file.

4 - Monitor indexing

Once the indexer is defined, it runs automatically when you submit the request. Depending on which cognitive skills you defined, indexing can take longer than you

expect. To find out whether the indexer is still running, use the `GetStatus` method.

C#

```
private static void CheckIndexerOverallStatus(SearchIndexerClient
indexerClient, SearchIndexer indexer)
{
    try
    {
        var demoIndexerExecutionInfo =
indexerClient.GetIndexerStatus(indexer.Name);

        switch (demoIndexerExecutionInfo.Value.Status)
        {
            case IndexerStatus.Error:
                ExitProgram("Indexer has error status. Check the Azure
Portal to further understand the error.");
                break;
            case IndexerStatus.Running:
                Console.WriteLine("Indexer is running");
                break;
            case IndexerStatus.Unknown:
                Console.WriteLine("Indexer status is unknown");
                break;
            default:
                Console.WriteLine("No indexer information");
                break;
        }
    }
    catch (RequestFailedException ex)
    {
        Console.WriteLine("Failed to get indexer overall status\n Exception
message: {0}\n", ex.Message);
    }
}
```

`demoIndexerExecutionInfo` represents the current status and execution history of an indexer.

Warnings are common with some source file and skill combinations and do not always indicate a problem. In this tutorial, the warnings are benign (for example, no text inputs from the JPEG files).

Add the following lines to `Main`.

C#

```
// Check indexer overall status
Console.WriteLine("Check the indexer overall status...");
CheckIndexerOverallStatus(indexerClient, demoIndexer);
```

5 - Search

In Azure Cognitive Search tutorial console apps, we typically add a 2-second delay before running queries that return results, but because enrichment takes several minutes to complete, we'll close the console app and use another approach instead.

The easiest option is Search explorer in the portal. You can first run an empty query that returns all documents, or a more targeted search that returns new field content created by the pipeline.

1. In Azure portal, in the search Overview page, select **Indexes**.
2. Find `demoindex` in the list. It should have 14 documents. If the document count is zero, the indexer is either still running or the page hasn't been refreshed yet.
3. Select `demoindex`. Search explorer is the first tab.
4. Content is searchable as soon as the first document is loaded. To verify content exists, run an unspecified query by clicking **Search**. This query returns all currently indexed documents, giving you an idea of what the index contains.
5. Next, paste in the following string for more manageable results: `search=*&$select=id, languageCode, organizations`

Reset and rerun

In the early experimental stages of development, the most practical approach for design iteration is to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

The sample code for this tutorial checks for existing objects and deletes them so that you can rerun your code. You can also use the portal to delete indexes, indexers, data sources, and skillsets.

Takeaways

This tutorial demonstrated the basic steps for building an enriched indexing pipeline through the creation of component parts: a data source, skillset, index, and indexer.

Built-in skills were introduced, along with skillset definition and the mechanics of chaining skills together through inputs and outputs. You also learned that

`outputFieldMappings` in the indexer definition is required for routing enriched values from the pipeline into a searchable index on an Azure Cognitive Search service.

Finally, you learned how to test results and reset the system for further iterations. You learned that issuing queries against the index returns the output created by the enriched indexing pipeline. You also learned how to check indexer status, and which objects to delete before rerunning a pipeline.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with all of the objects in an AI enrichment pipeline, let's take a closer look at skillset definitions and individual skills.

How to create a skillset

Tutorial: Use REST and AI to generate searchable content from Azure blobs

Article • 03/24/2023 • 16 minutes to read

If you have unstructured text or images in Azure Blob Storage, an AI enrichment pipeline can extract information and create new content for full-text search or knowledge mining scenarios.

In this REST tutorial, you'll learn how to:

- ✓ Set up a development environment.
- ✓ Define a pipeline that uses OCR, language detection, entity recognition, and key phrase extraction.
- ✓ Execute the pipeline to invoke transformations, and to create and load a search index.
- ✓ Explore results using full text search and a rich query syntax.

If you don't have an Azure subscription, open a free account before you begin.

Overview

This tutorial uses Postman and the Azure Cognitive Search REST APIs to create a data source, index, indexer, and skillset.

The indexer connects to Azure Blob Storage and retrieves the content, which you must load in advance. The indexer then invokes a skillset for specialized processing, and ingests the enriched content into a search index.

The skillset is attached to an indexer. It uses built-in skills from Microsoft to find and extract information. Steps in the pipeline include Optical Character Recognition (OCR) on images, language detection, key phrase extraction, and entity recognition (organizations, locations, people). New information created by the pipeline is stored in new fields in an index. Once the index is populated, you can use those fields in queries, facets, and filters.

Prerequisites

- Postman app
- Azure Storage
- Azure Cognitive Search

- Sample data

> **Note**
>
> You can use the free service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

Download files

The sample data consists of 14 files of mixed content type that you'll upload to Azure Blob Storage in a later step.

1. Get the files from azure-search-sample-data/ai-enrichment-mixed-media/ and copy them to your local computer.
2. Next, get the source code, a Postman collection file, for this tutorial. Source code can be found at azure-search-postman-samples/tree/master/Tutorial.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, Cognitive Services on the backend for AI enrichment, and Azure Blob Storage to provide the data. This tutorial stays under the free allocation of 20 transactions per indexer per day on Cognitive Services, so the only services you need to create are search and storage.

If possible, create both in the same region and resource group for proximity and manageability. In practice, your Azure Storage account can be in any region.

Start with Azure Storage

1. Sign in to the Azure portal and select + **Create Resource**.
2. Search for *storage account* and select Microsoft's Storage Account offering.

3. In the Basics tab, the following items are required. Accept the defaults for everything else.

    - **Resource group**. Select an existing one or create a new one, but use the same group for all services so that you can manage them collectively.
    - **Storage account name**. If you think you might have multiple resources of the same type, use the name to disambiguate by type and region, for example *blobstoragewestus*.
    - **Location**. If possible, choose the same location used for Azure Cognitive Search and Cognitive Services. A single location voids bandwidth charges.
    - **Account Kind**. Choose the default, *StorageV2 (general purpose v2)*.

4. Select **Review + Create** to create the service.

5. Once it's created, select **Go to the resource** to open the Overview page.

6. Select **Blobs** service.

7. Select **+ Container** to create a container and name it *cog-search-demo*.

8. Select *cog-search-demo* and then select **Upload** to open the folder where you saved the download files. Select all of the files. Select **Upload**.

Name	Date modified	Type	Size
facts-about-microsoft.html	12/8/2021 12:16 PM	Chrome HTML Document	138 KB
guthrie.jpg	12/23/2019 8:52 AM	JPG File	46 KB
redshirt.jpg	12/23/2019 8:52 AM	JPG File	67 KB
satyanadellalinux.jpg	12/23/2019 8:52 AM	JPG File	108 KB
Cognitive Services and Bots (spanish).pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	3,884 KB
increase_green_energy_ one_rooftop_at_a_time.pdf	12/8/2021 12:35 PM	Microsoft Edge PDF Document	499 KB
MSFT_cloud_architecture_contoso.pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	1,805 KB
Cognitive Searvices and Content Intelligence.pptx	12/23/2019 8:52 AM	Microsoft PowerPoint Presenta...	11,231 KB
Mesh_for_Microsoft_Teams.docx	12/8/2021 12:23 PM	Microsoft Word Document	4,137 KB
5074.clip_image002_6FE27E85.png	12/23/2019 8:52 AM	PNG File	472 KB
create-search-collect-info.png	12/23/2019 8:52 AM	PNG File	57 KB
create-search-service.png	12/23/2019 8:52 AM	PNG File	26 KB
create-service-full-portal.png	12/23/2019 8:52 AM	PNG File	67 KB
satyasletter.txt	12/23/2019 8:52 AM	Text Document	6 KB

9. Before you leave Azure Storage, get a connection string so that you can formulate a connection in Azure Cognitive Search.

    a. Browse back to the Overview page of your storage account (we used *blobstragewestus* as an example).

    b. In the left navigation pane, select **Access keys** and copy one of the connection strings.

The connection string is a URL similar to the following example:

```HTTP
DefaultEndpointsProtocol=https;AccountName=cogsrchdemostorage;AccountKey=<your account key>;EndpointSuffix=core.windows.net
```

10. Save the connection string to Notepad. You'll need it later when setting up the data source connection.

Cognitive Services

AI enrichment is backed by Cognitive Services, including Language service and Computer Vision for natural language and image processing. If your objective was to complete an actual prototype or project, you would at this point provision Cognitive Services (in the same region as Azure Cognitive Search) so that you can attach it to a skillset.

For this exercise, however, you can skip resource provisioning because Azure Cognitive Search can connect to Cognitive Services execute 20 transactions per indexer run, free of charge. Since this tutorial uses 14 transactions, the free allocation is sufficient. For larger projects, plan on provisioning Cognitive Services at the pay-as-you-go S0 tier.

Azure Cognitive Search

The third component is Azure Cognitive Search, which you can create in the portal or find an existing search service in your subscription.

You can use the Free tier to complete this walkthrough.

Copy an admin api-key and URL for Azure Cognitive Search

To interact with your Azure Cognitive Search service you'll need the service URL and an access key.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the name of your search service. You can confirm your service name by reviewing the endpoint URL. If your endpoint URL were `https://mydemo.search.windows.net`, your service name would be `mydemo`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. You can copy either the primary or secondary key.

All HTTP requests to a search service require an API key. A valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Set up Postman

1. Start Postman, import the collection, and set up the environment variables. If you're unfamiliar with this tool, see Explore Azure Cognitive Search REST APIs.

2. You'll need to provide a search service name, an admin API key, an index name, a connection string to your Azure Storage account, and the container name.

The request methods used in this collection are **PUT** and **GET**. You'll use the methods to create a data source, a skillset, an index, and an indexer.

3 - Create the pipeline

In Azure Cognitive Search, enrichment occurs during indexing (or data ingestion). This part of the walkthrough creates four objects: data source, index definition, skillset, indexer.

Step 1: Create a data source

Call Create Data Source to set the connection string to the Blob container containing the sample data files.

1. Select the "Create a data source" request.

2. The body of the request is JSON and includes properties of an indexer data source object. The connection string includes credentials for accessing the service.

JSON

```
{
    "description" : "Demo files to demonstrate cognitive search capabilities.",
    "type" : "azureblob",
    "credentials" : {
       "connectionString": "{{azure_storage_connection_string}}"
```

```
    },
  "container" : {
    "name" : "{{container_name}}"
  }
}
```

3. Send the request. You should see a status code of 201 confirming success.

If you got a 403 or 404 error, check the search admin API key and the Azure Storage connection string.

Step 2: Create a skillset

Call Create Skillset to specify which enrichment steps are applied to your content.

1. Select the "Create a skillset" request.

2. The body of the request specifies the following built-in skills:

Skill	Description
Optical Character Recognition	Recognizes text and numbers in image files.
Text Merge	Creates "merged content" that recombines previously separated content, useful for documents with embedded images (PDF, DOCX, and so forth). Images and text are separated during the document cracking phase. The merge skill recombines them by inserting any recognized text, image captions, or tags created during enrichment into the same location where the image was extracted from in the document. When you're working with merged content in a skillset, this node will be inclusive of all text in the document, including text-only documents that never undergo OCR or image analysis.
Language Detection	Detects the language and outputs either a language name or code. In multilingual data sets, a language field can be useful for filters.
Entity Recognition	Extracts the names of people, organizations, and locations from merged content.
Text Split	Breaks large merged content into smaller chunks before calling the key phrase extraction skill. Key phrase extraction accepts inputs of 50,000 characters or less. A few of the sample files need splitting up to fit within this limit.
Key Phrase Extraction	Pulls out the top key phrases.

Each skill executes on the content of the document. During processing, Azure Cognitive Search cracks each document to read content from different file formats. Found text originating in the source file is placed into a generated `content` field, one for each document. As such, the input becomes `"/document/content"`.

For key phrase extraction, because we use the text splitter skill to break larger files into pages, the context for the key phrase extraction skill is `"document/pages/*"` (for each page in the document) instead of `"/document/content"`.

JSON

```
{
  "description": "Apply OCR, detect language, extract entities, and
extract key-phrases.",
  "cognitiveServices": null,
  "skills":
  [
    {
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
      "context": "/document/normalized_images/*",
      "defaultLanguageCode": "en",
      "detectOrientation": true,
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [
        {
          "name": "text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
      "description": "Create merged_text, which includes all the
textual representation of each image inserted at the right location in
the content field. This is useful for PDF and other file formats that
supported embedded images.",
      "context": "/document",
      "insertPreTag": " ",
      "insertPostTag": " ",
      "inputs": [
        {
          "name":"text",
          "source": "/document/content"
        },
        {
          "name": "itemsToInsert",
          "source": "/document/normalized_images/*/text"
        },
```

```
        {
          "name":"offsets",
          "source": "/document/normalized_images/*/contentOffset"
        }
      ],
      "outputs": [
        {
          "name": "mergedText",
          "targetName" : "merged_text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
      "textSplitMode": "pages",
      "maximumPageLength": 4000,
      "defaultLanguageCode": "en",
      "context": "/document",
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_text"
        }
      ],
      "outputs": [
        {
          "name": "textItems",
          "targetName": "pages"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
      "description": "If you have multilingual content, adding a
language code is useful for filtering",
      "context": "/document",
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_text"
        }
      ],
      "outputs": [
        {
          "name": "languageName",
          "targetName": "language"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
      "context": "/document/pages/*",
      "inputs": [
        {
          "name": "text",
```

```
          "source": "/document/pages/*"
        }
      ],
      "outputs": [
        {
          "name": "keyPhrases",
          "targetName": "keyPhrases"
        }
      ]
    },
    {
      "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "categories": ["Organization"],
      "context": "/document",
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_text"
        }
      ],
      "outputs": [
        {
          "name": "organizations",
          "targetName": "organizations"
        }
      ]
    },
    {
      "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "categories": ["Location"],
      "context": "/document",
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_text"
        }
      ],
      "outputs": [
        {
          "name": "locations",
          "targetName": "locations"
        }
      ]
    },
    {
      "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "categories": ["Person"],
      "context": "/document",
      "inputs": [
        {
          "name": "text",
          "source": "/document/merged_text"
```

```
        }
      ],
      "outputs": [
        {
          "name": "persons",
          "targetName": "persons"
        }
      ]
    }
  ]
}
```

A graphical representation of a portion of the skillset is shown below.

3. Send the request. Postman should return a status code of 201 confirming success.

> ⓘ **Note**
>
> Outputs can be mapped to an index, used as input to a downstream skill, or both as is the case with language code. In the index, a language code is useful for filtering. For more information about skillset fundamentals, see **How to define a skillset**.

Step 3: Create an index

Call Create Index to provide the schema used to create inverted indexes and other constructs in Azure Cognitive Search. The largest component of an index is the fields collection, where data type and attributes determine content and behavior in Azure Cognitive Search.

1. Select the "Create an index" request.

2. The body of the request defines the schema of the search index. A fields collection requires one field to be designated as the key. For blob content, this field is often the "metadata_storage_path" that uniquely identifies each blob in the container.

   In this schema, the "text" field receives OCR output, "content" receives merged output, "language" receives language detection output. Key phrases, entities, and several fields lifted from blob storage comprise the remaining entries.

JSON

```
{
  "fields": [
    {
      "name": "text",
      "type": "Collection(Edm.String)",
      "searchable": true,
      "sortable": false,
      "filterable": true,
      "facetable": false
    },
    {
      "name": "content",
      "type": "Edm.String",
      "searchable": true,
      "sortable": false,
      "filterable": false,
      "facetable": false
    },
    {
      "name": "language",
      "type": "Edm.String",
      "searchable": false,
      "sortable": true,
      "filterable": true,
      "facetable": false
    },
    {
      "name": "keyPhrases",
      "type": "Collection(Edm.String)",
      "searchable": true,
      "sortable": false,
      "filterable": true,
      "facetable": true
    },
    {
      "name": "organizations",
      "type": "Collection(Edm.String)",
      "searchable": true,
      "sortable": false,
      "filterable": true,
      "facetable": true
    },
    {
      "name": "persons",
      "type": "Collection(Edm.String)",
      "searchable": true,
      "sortable": false,
      "filterable": true,
      "facetable": true
    },
    {
      "name": "locations",
```

```
        "type": "Collection(Edm.String)",
        "searchable": true,
        "sortable": false,
        "filterable": true,
        "facetable": true
      },
      {
        "name": "metadata_storage_path",
        "type": "Edm.String",
        "key": true,
        "searchable": true,
        "sortable": false,
        "filterable": false,
        "facetable": false
      },
      {
        "name": "metadata_storage_name",
        "type": "Edm.String",
        "searchable": true,
        "sortable": false,
        "filterable": false,
        "facetable": false
      }
    ]
  }
```

3. Send the request. Postman should return a status code of 201 confirming success.

Step 4: Create and run an indexer

Call Create Indexer to drive the pipeline. The three components you have created thus far (data source, skillset, index) are inputs to an indexer. Creating the indexer on Azure Cognitive Search is the event that puts the entire pipeline into motion.

1. Select the "Create an indexer" request.

2. The body of the request includes references to the previous objects, configuration properties required for image processing, and two types of field mappings.

   `"fieldMappings"` are processed before the skillset, sending content from the data source to target fields in an index. You'll use field mappings to send existing, unmodified content to the index. If field names and types are the same at both ends, no mapping is required.

   `"outputFieldMappings"` are for fields created by skills, after skillset execution. The references to `sourceFieldName` in `outputFieldMappings` don't exist until document cracking or enrichment creates them. The `targetFieldName` is a field in an index, defined in the index schema.

JSON

```
{
  "dataSourceName" : "{{index_name}}-datasource",
  "targetIndexName" : "{{index_name}}",
  "skillsetName" : "{{index_name}}-skillset",
  "fieldMappings" : [
        {
          "sourceFieldName" : "metadata_storage_path",
          "targetFieldName" : "metadata_storage_path",
          "mappingFunction" : { "name" : "base64Encode" }
        },
        {
            "sourceFieldName": "metadata_storage_name",
            "targetFieldName": "metadata_storage_name"
        }
   ],
  "outputFieldMappings" :
    [
        {
            "sourceFieldName": "/document/merged_text",
            "targetFieldName": "content"
        },
        {
            "sourceFieldName" : "/document/normalized_images/*/text",
            "targetFieldName" : "text"
        },
        {
          "sourceFieldName" : "/document/organizations",
          "targetFieldName" : "organizations"
        },
        {
            "sourceFieldName": "/document/language",
            "targetFieldName": "language"
        },
        {
          "sourceFieldName" : "/document/persons",
          "targetFieldName" : "persons"
        },
        {
          "sourceFieldName" : "/document/locations",
          "targetFieldName" : "locations"
        },
        {
          "sourceFieldName" : "/document/pages/*/keyPhrases/*",
          "targetFieldName" : "keyPhrases"
        }
    ],
  "parameters":
  {
    "batchSize": 1,
    "maxFailedItems":-1,
    "maxFailedItemsPerBatch":-1,
    "configuration":
```

```
      {
          "dataToExtract": "contentAndMetadata",
          "imageAction": "generateNormalizedImages"
      }
    }
  }
```

3. Send the request. Postman should return a status code of 201 confirming successful processing.

   Expect this step to take several minutes to complete. Even though the data set is small, analytical skills are computation-intensive.

> **Note**
>
> Creating an indexer invokes the pipeline. If there are problems reaching the data, mapping inputs and outputs, or order of operations, they appear at this stage. To re-run the pipeline with code or script changes, you might need to drop objects first. For more information, see **Reset and re-run**.

About indexer parameters

The script sets `"maxFailedItems"` to -1, which instructs the indexing engine to ignore errors during data import. This is acceptable because there are so few documents in the demo data source. For a larger data source, you would set the value to greater than 0.

The `"dataToExtract":"contentAndMetadata"` statement tells the indexer to automatically extract the values from the blob's content property and the metadata of each object.

When content is extracted, you can set `imageAction` to extract text from images found in the data source. The `"imageAction":"generateNormalizedImages"` configuration, combined with the OCR Skill and Text Merge Skill, tells the indexer to extract text from the images (for example, the word "stop" from a traffic Stop sign), and embed it as part of the content field. This behavior applies to both embedded images (think of an image inside a PDF) and standalone image files, for instance a JPG file.

4 - Monitor indexing

Indexing and enrichment commence as soon as you submit the Create Indexer request. Depending on which cognitive skills you defined, indexing can take a while.

To find out whether the indexer is still running, call Get Indexer Status to check the indexer status.

1. Select and then send the "Check indexer status" request.

2. Review the response to learn whether the indexer is running, or to view error and warning information.

Warnings are common in some scenarios and do not always indicate a problem. For example, if a blob container includes image files, and the pipeline doesn't handle images, you'll get a warning stating that images were not processed.

In this sample, there is a PNG file that contains no text. All five of the text-based skills (language detection, entity recognition of locations, organizations, people, and key phrase extraction) fail to execute on this file. The resulting notification shows up in execution history.

5 - Search

Now that you've created an index that contains AI-generated content, call Search Documents to run some queries to see the results.

Recall that we started with blob content, where the entire document is packaged into a single `content` field. You can search this field and find matches to your queries.

1. Open the "Search" request and run it to get your first look at index content. This request is an empty search ("search=*") so it will return content for each of the 14 documents. The $select parameter constrains results to the file name, the language name, and one of the recognized entities.

   HTTP

   ```
   GET /indexes//{{index_name}}/docs?search=*&$select=metadata_storage_name,language,organizations&$count=true&api-version=2020-06-30
   ```

2. Revise the previous query to search for "creating boundaryless opportunities". This phrase was obtained through OCR of an embedded image file in a PDF document. Include "highlight" to apply formatting on matching terms in densely populated fields.

   HTTP

```
GET /indexes//{{index_name}}/docs?search=creating boundaryless
opportunities&$select=content&highlight=content&$count=true&api-
version=2020-06-30
```

3. For the next query, apply a filter. Recall that the language field and all entity fields are filterable.

HTTP

```
GET /indexes/{{index_name}}/docs?
search=*&$filter=organizations/any(organizations: organizations eq
'Microsoft')&$select=metadata_storage_name,organizations&$count=true&ap
i-version=2020-06-30
```

These queries illustrate a few of the ways you can work with query syntax and filters on new fields created by cognitive search. For more query examples, see Examples in Search Documents REST API, Simple syntax query examples, and Full Lucene query examples.

Reset and rerun

During early stages of development, iteration over the design is common. You will most likely delete and rebuild the same objects frequently.

If you use the portal for deletion, and delete the indexer first, the portal will prompt you to delete the associated objects.

Alternatively, you can use **DELETE** and provide URLs to each object. The following command deletes an indexer.

```HTTP
DELETE https://[YOUR-SERVICE-NAME].search.windows.net/indexers/cog-search-demo-idxr?api-version=2020-06-30
```

Status code 204 is returned on successful deletion.

Takeaways

This tutorial demonstrates the basic steps for building an enriched indexing pipeline through the creation of component parts: a data source, skillset, index, and indexer.

Built-in skills were introduced, along with skillset definition and the mechanics of chaining skills together through inputs and outputs. You also learned that `outputFieldMappings` in the indexer definition is required for routing enriched values from the pipeline into a searchable index on an Azure Cognitive Search service.

Finally, you learned how to test results and reset the system for further iterations. You learned that issuing queries against the index returns the output created by the enriched indexing pipeline.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with all of the objects in an AI enrichment pipeline, let's take a closer look at skillset definitions and individual skills.

How to create a skillset

Tutorial: Use Python and AI to generate searchable content from Azure blobs

Article • 08/26/2022 • 16 minutes to read

If you have unstructured text or images in Azure Blob Storage, an AI enrichment pipeline can extract information and create new content for full-text search or knowledge mining scenarios.

In this Python tutorial, you'll learn how to:

- ✓ Set up a development environment
- ✓ Define a pipeline that uses OCR, language detection, and entity and key phrase recognition.
- ✓ Execute the pipeline to invoke transformations, and to create and load a search index.
- ✓ Explore results using full text search and a rich query syntax.

If you don't have an Azure subscription, open a free account before you begin.

Overview

This tutorial uses Python and the Search REST APIs to create a data source, index, indexer, and skillset.

The indexer retrieves sample data in a blob container that's specified in the data source object, and sends all enriched content to a search index.

The skillset is attached to the indexer. It uses built-in skills from Microsoft to find and extract information. Steps in the pipeline include Optical Character Recognition (OCR) on images, language detection on text, key phrase extraction, and entity recognition (organizations). New information created by the pipeline is stored in new fields in an index. Once the index is populated, you can use the fields in queries, facets, and filters.

Prerequisites

- Visual Studio Code with the Python extension and Python 3.7 or later
- Azure Storage
- Azure Cognitive Search

ⓘ **Note**

You can use the free search service for this tutorial. A free search service limits you to three indexes, three indexers, and three data sources. This tutorial creates one of each. Before starting, make sure you have room on your service to accept the new resources.

Download files

The sample data consists of 14 files of mixed content type that you'll upload to Azure Blob Storage in a later step.

1. Open this OneDrive folder and on the top-left corner, select **Download** to copy the files to your computer.

2. Right-click the zip file and select **Extract All**. There are 14 files of various types.

Optionally, you can also download the source code for this tutorial. Source code can be found at https://github.com/Azure-Samples/azure-search-python-samples/tree/master/Tutorial-AI-Enrichment.

1 - Create services

This tutorial uses Azure Cognitive Search for indexing and queries, Cognitive Services on the backend for AI enrichment, and Azure Blob Storage to provide the data. This tutorial stays under the Cognitive Search free allocation of 20 transactions per indexer per day on Cognitive Services, so the only services you need to create are search and storage.

If possible, create both in the same region and resource group for proximity and manageability. In practice, your Azure Storage account can be in any region.

Start with Azure Storage

1. Sign in to the Azure portal and select + **Create Resource**.

2. Search for *storage account* and select Microsoft's Storage Account offering.

3. In the Basics tab, the following items are required. Accept the defaults for everything else.

- **Resource group**. Select an existing one or create a new one, but use the same group for all services so that you can manage them collectively.
- **Storage account name**. If you think you might have multiple resources of the same type, use the name to disambiguate by type and region, for example *blobstoragewestus*.
- **Location**. If possible, choose the same location used for Azure Cognitive Search and Cognitive Services. A single location voids bandwidth charges.
- **Account Kind**. Choose the default, *StorageV2 (general purpose v2)*.

4. Select **Review + Create** to create the service.
5. Once it's created, select **Go to the resource** to open the Overview page.
6. Select **Blobs** service.
7. Select **+ Container** to create a container and name it *cog-search-demo*.
8. Select *cog-search-demo* and then select **Upload** to open the folder where you saved the download files. Select all of the non-image files. You should have 14 files. Select **OK** to upload.

Name	Date modified	Type	Size
facts-about-microsoft.html	12/8/2021 12:16 PM	Chrome HTML Document	138 KB
guthrie.jpg	12/23/2019 8:52 AM	JPG File	46 KB
redshirt.jpg	12/23/2019 8:52 AM	JPG File	67 KB
satyanadellalinux.jpg	12/23/2019 8:52 AM	JPG File	108 KB
Cognitive Services and Bots (spanish).pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	3,884 KB
increase_green_energy_ one_rooftop_at_a_time.pdf	12/8/2021 12:35 PM	Microsoft Edge PDF Document	499 KB
MSFT_cloud_architecture_contoso.pdf	12/23/2019 8:52 AM	Microsoft Edge PDF Document	1,805 KB
Cognitive Searvices and Content Intelligence.pptx	12/23/2019 8:52 AM	Microsoft PowerPoint Presenta...	11,231 KB
Mesh_for_Microsoft_Teams.docx	12/8/2021 12:23 PM	Microsoft Word Document	4,137 KB
5074.clip_image002_6FE27E85.png	12/23/2019 8:52 AM	PNG File	472 KB
create-search-collect-info.png	12/23/2019 8:52 AM	PNG File	57 KB
create-search-service.png	12/23/2019 8:52 AM	PNG File	26 KB
create-service-full-portal.png	12/23/2019 8:52 AM	PNG File	67 KB
satyasletter.txt	12/23/2019 8:52 AM	Text Document	6 KB

9. Before you leave Azure Storage, get a connection string so that you can formulate a connection in Azure Cognitive Search.

   a. Browse back to the Overview page of your storage account (we used *blobstragewestus* as an example).

   b. In the left navigation pane, select **Access keys** and copy one of the connection strings.

   The connection string is a URL similar to the following example:

```HTTP
DefaultEndpointsProtocol=https;AccountName=
<storageaccountname>;AccountKey=<your account
key>;EndpointSuffix=core.windows.net
```

10. Save the connection string to Notepad. You'll need it later when setting up the data source connection.

Cognitive Services

AI enrichment is backed by Cognitive Services, including Language service and Computer Vision for natural language and image processing. If your objective was to complete an actual prototype or project, you would at this point provision Cognitive Services (in the same region as Azure Cognitive Search) so that you can attach it to indexing operations.

Since this tutorial only uses 14 transactions, you can skip resource provisioning because Azure Cognitive Search can connect to Cognitive Services for 20 free transactions per indexer run. For larger projects, plan on provisioning Cognitive Services at the pay-as-you-go S0 tier. For more information, see Attach Cognitive Services.

Azure Cognitive Search

The third component is Azure Cognitive Search, which you can create in the portal or find an existing search service in your subscription.

You can use the Free tier to complete this tutorial.

Copy an admin api-key and URL for Azure Cognitive Search

To send requests to your Azure Cognitive Search service, you'll need the service URL and an access key.

1. Sign in to the Azure portal, and in your search service **Overview** page, get the name of your search service. You can confirm your service name by reviewing the endpoint URL. If your endpoint URL is `https://mydemo.search.windows.net`, your service name would be `mydemo`.

2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to

roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key in the header of every request sent to your service. A valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

2 - Start a notebook

Use Visual Studio Code with the Python extension to create a new notebook. Press F1 to open the command palette and then search for "Create: New Jupyter Notebook".

Alternatively, if you downloaded the notebook from Azure-Search-python-samples repo, you can open it in Visual Studio Code.

In your notebook, create a new cell and add this script. It loads the libraries used for working with JSON and formulating HTTP requests.

Python

```python
import json
import requests
from pprint import pprint
```

In the next cell, define the names for the data source, index, indexer, and skillset. Run this script to set up the names for this tutorial.

Python

```
# Define the names for the data source, skillset, index and indexer
datasource_name = "cogsrch-py-datasource"
skillset_name = "cogsrch-py-skillset"
index_name = "cogsrch-py-index"
indexer_name = "cogsrch-py-indexer"
```

In a third cell, paste the following script, replacing the placeholders for your search service (YOUR-SEARCH-SERVICE-NAME) and admin API key (YOUR-ADMIN-API-KEY), and then run it to set up the search service endpoint.

Python

```
# Setup the endpoint
endpoint = 'https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/'
headers = {'Content-Type': 'application/json',
           'api-key': '<YOUR-ADMIN-API-KEY>'}
params = {
    'api-version': '2020-06-30'
}
```

3 - Create the pipeline

In Azure Cognitive Search, AI processing occurs during indexing (or data ingestion). This part of the tutorial creates four objects: data source, index definition, skillset, indexer.

Step 1: Create a data source

A data source object provides the connection string to the Blob container containing the sample data files.

In the following script, replace the placeholder YOUR-BLOB-RESOURCE-CONNECTION-STRING with the connection string for the blob you created in the previous step. Replace the placeholder YOUR-BLOB-CONTAINER-NAME with the name of your container. Then, run the script to create a data source named `cogsrch-py-datasource`.

Python

```
# Create a data source
datasourceConnectionString = "<YOUR-BLOB-RESOURCE-CONNECTION-STRING>"
datasource_payload = {
    "name": datasource_name,
    "description": "Demo files to demonstrate cognitive search
capabilities.",
    "type": "azureblob",
    "credentials": {
```

```
        "connectionString": datasourceConnectionString
    },
    "container": {
        "name": "<YOUR-BLOB-CONTAINER-NAME>"
    }
}
r = requests.put(endpoint + "/datasources/" + datasource_name,
                 data=json.dumps(datasource_payload), headers=headers,
params=params)
print(r.status_code)
```

The request should return a status code of 201 confirming success.

In the Azure portal, on the search service dashboard page, verify that the cogsrch-py-datasource appears in the **Data sources** list. Select **Refresh** to update the page.

Step 2: Create a skillset

In this step, you'll define a set of enrichment steps using built-in cognitive skills from Microsoft:

- Entity Recognition for extracting the names of organizations from content in the blob container.
- Language Detection to identify the content's language.
- Text Split to break large content into smaller chunks before calling the key phrase extraction skill. Key phrase extraction accepts inputs of 50,000 characters or less. A few of the sample files need splitting up to fit within this limit.
- Key Phrase Extraction to pull out the top key phrases.

Run the following script to create a skillset called `cogsrch-py-skillset`.

Python

```
# Create a skillset
skillset_payload = {
    "name": skillset_name,
    "description":
    "Extract entities, detect language and extract key-phrases",
    "skills":
    [
        {
            "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
            "categories": ["Organization"],
            "defaultLanguageCode": "en",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/content"
                }
            ],
            "outputs": [
                {
                    "name": "organizations",
                    "targetName": "organizations"
                }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/content"
                }
            ],
            "outputs": [
                {
                    "name": "languageCode",
                    "targetName": "languageCode"
                }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
            "textSplitMode": "pages",
            "maximumPageLength": 4000,
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/content"
                },
                {
                    "name": "languageCode",
                    "source": "/document/languageCode"
                }
            ],
```

```
            "outputs": [
                {
                    "name": "textItems",
                    "targetName": "pages"
                }
            ]
        },
        {
            "@odata.type":
"#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
            "context": "/document/pages/*",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/pages/*"
                },
                {
                    "name": "languageCode",
                    "source": "/document/languageCode"
                }
            ],
            "outputs": [
                {
                    "name": "keyPhrases",
                    "targetName": "keyPhrases"
                }
            ]
        }
    ]
}

r = requests.put(endpoint + "/skillsets/" + skillset_name,
                 data=json.dumps(skillset_payload), headers=headers,
params=params)
print(r.status_code)
```

The request should return a status code of 201 confirming success.

The key phrase extraction skill is applied for each page. By setting the context to `"document/pages/*"`, you run this enricher for each member of the document/pages array (for each page in the document).

Each skill executes on the content of the document. During processing, Azure Cognitive Search cracks each document to read content from different file formats. Text found in the source file is placed into a `content` field, one for each document. Therefore, set the input as `"/document/content"`.

A graphical representation of a portion of the skillset is shown below.

Outputs can be mapped to an index, used as input to a downstream skill, or both, as is the case with language code. In the index, a language code is useful for filtering. As an input, language code is used by text analysis skills to inform the linguistic rules around word breaking.

For more information about skillset fundamentals, see How to define a skillset.

Step 3: Create an index

In this section, you define the index schema by specifying the fields to include in the searchable index, and setting the search attributes for each field. Fields have a type and can take attributes that determine how the field is used (searchable, sortable, and so forth). Field names in an index aren't required to identically match the field names in the source. In a later step, you add field mappings in an indexer to connect source-destination fields. For this step, define the index using field naming conventions pertinent to your search application.

This exercise uses the following fields and field types:

field-names:	**ID**	**content**	**languageCode**	**keyPhrases**	**organizations**
field-types:	Edm.String	Edm.String	Edm.String	List<Edm.String>	List<Edm.String>

Run this script to create the index named `cogsrch-py-index`.

Python

```python
# Create an index
index_payload = {
    "name": index_name,
    "fields": [
        {
            "name": "id",
            "type": "Edm.String",
            "key": "true",
            "searchable": "true",
            "filterable": "false",
            "facetable": "false",
            "sortable": "true"
        },
```

```
        {
            "name": "content",
            "type": "Edm.String",
            "sortable": "false",
            "searchable": "true",
            "filterable": "false",
            "facetable": "false"
        },
        {
            "name": "languageCode",
            "type": "Edm.String",
            "searchable": "true",
            "filterable": "false",
            "facetable": "false"
        },
        {
            "name": "keyPhrases",
            "type": "Collection(Edm.String)",
            "searchable": "true",
            "filterable": "false",
            "facetable": "false"
        },
        {
            "name": "organizations",
            "type": "Collection(Edm.String)",
            "searchable": "true",
            "sortable": "false",
            "filterable": "false",
            "facetable": "false"
        }
    ]
}

r = requests.put(endpoint + "/indexes/" + index_name,
                 data=json.dumps(index_payload), headers=headers,
params=params)
print(r.status_code)
```

The request should return a status code of 201 confirming success.

To learn more about defining an index, see Create Index (REST API).

Step 4: Create and run an indexer

An Indexer drives the pipeline. The three components you've created thus far (data source, skillset, index) are inputs to an indexer. Creating the indexer on Azure Cognitive Search is the event that puts the entire pipeline into motion.

To tie these objects together in an indexer, you must define field mappings.

- The `"fieldMappings"` are processed before the skillset, mapping source fields from the data source to target fields in an index. If field names and types are the same at both ends, no mapping is required.
- The `"outputFieldMappings"` are processed after the skillset, referencing `"sourceFieldNames"` that don't exist until document cracking or enrichment creates them. The `"targetFieldName"` is a field in an index.

Besides hooking up inputs to outputs, you can also use field mappings to flatten data structures. For more information, see How to map enriched fields to a searchable index.

Run this script to create an indexer named `cogsrch-py-indexer`.

Python

```python
# Create an indexer
indexer_payload = {
    "name": indexer_name,
    "dataSourceName": datasource_name,
    "targetIndexName": index_name,
    "skillsetName": skillset_name,
    "fieldMappings": [
        {
            "sourceFieldName": "metadata_storage_path",
            "targetFieldName": "id",
            "mappingFunction":
            {"name": "base64Encode"}
        },
        {
            "sourceFieldName": "content",
            "targetFieldName": "content"
        }
    ],
    "outputFieldMappings":
    [
        {
            "sourceFieldName": "/document/organizations",
            "targetFieldName": "organizations"
        },
        {
            "sourceFieldName": "/document/pages/*/keyPhrases/*",
            "targetFieldName": "keyPhrases"
        },
        {
            "sourceFieldName": "/document/languageCode",
            "targetFieldName": "languageCode"
        }
    ],
    "parameters":
    {
        "maxFailedItems": -1,
```

```
        "maxFailedItemsPerBatch": -1,
        "configuration":
        {
            "dataToExtract": "contentAndMetadata",
            "imageAction": "generateNormalizedImages"
        }
    }
}

r = requests.put(endpoint + "/indexers/" + indexer_name,
                 data=json.dumps(indexer_payload), headers=headers,
params=params)
print(r.status_code)
```

The request should return a status code of 201 soon, however, the processing can take several minutes to complete. Although the data set is small, analytical skills, such as image analysis, are computationally intensive and take time.

You can monitor indexer status to determine when the indexer is running or finished.

> **Tip**
>
> Creating an indexer invokes the pipeline. If there is a problem accessing the data, mapping inputs and outputs, or with the order of operations, it will appear at this stage. To re-run the pipeline with code or script changes, you may need to delete objects first. For more information, see **Reset and re-run**.

About the request body

The script sets `"maxFailedItems"` to -1, which instructs the indexing engine to ignore errors during data import. This is useful because there are so few documents in the demo data source. For a larger data source, you would set the value to greater than 0.

Also notice the `"dataToExtract":"contentAndMetadata"` statement in the configuration parameters. This statement tells the indexer to extract the content from different file formats and the metadata related to each file.

When content is extracted, you can set `imageAction` to extract text from images found in the data source. The `"imageAction":"generateNormalizedImages"` configuration, combined with the OCR Skill and Text Merge Skill, tells the indexer to extract text from the images (for example, the word "stop" from a traffic Stop sign), and embed it as part of the content field. This behavior applies to both the images embedded in the documents (think of an image inside a PDF) and images found in the data source, for instance a JPG file.

4 - Monitor indexing

Once the indexer is defined, it runs automatically when you submit the request. Depending on which cognitive skills you defined, indexing can take longer than you expect. To find out whether the indexer processing is complete, run the following script.

Python

```python
# Get indexer status
r = requests.get(endpoint + "/indexers/" + indexer_name +
                  "/status", headers=headers, params=params)
pprint(json.dumps(r.json(), indent=1))
```

In the response, monitor the `"lastResult"` for its `"status"` and `"endTime"` values. Periodically run the script to check the status. When the indexer has completed, the status will be set to "success", an "endTime" will be specified, and the response will include any errors and warnings that occurred during enrichment.

```python
#Determine if the indexer is still running
r = requests.get(endpoint + "/indexers/" + indexer_name + "/status", headers=headers,params=params)
pprint(json.dumps(r.json(), indent=1))
```

```
('{\n'
 ' "@odata.context": '
 '"https://mysearchsvc.search.windows.net/$metadata#Microsoft.Azure.Search.V2019_05_06.IndexerExecutionInfo",\n'
 ' "name": "cogsrch-py-indexer",\n'
 ' "status": "running",\n'
 ' "lastResult": {\n'
 '  "status": "success",\n'
 '  "errorMessage": null,\n'
 '  "startTime": "2019-06-13T15:15:14.078Z",\n'
 '  "endTime": "2019-06-13T15:15:47.66Z",\n'
 '  "itemsProcessed": 14,\n'
 '  "itemsFailed": 0,\n'
 '  "initialTrackingState": "{\\r\\n  \\"lastFullEnumerationStartTime\\": '
 '\\"0001-01-01T00:00:00Z\\",\\r\\n  \\"lastAttemptedEnumerationStartTime\\": '
 '\\"0001-01-01T00:00:00Z\\",\\r\\n  \\"nameHighWaterMark\\": null\\r\\n}",\n'
 '  "finalTrackingState": '
 '"{\\"LastFullEnumerationStartTime\\":\\"2019-06-13T15:14:44.0784124+00:00\\",\\"LastAttemptedEnumerationStartTime\\":\\"201
9-06-13T15:14:44.0784124+00:00\\",\\"NameHighWaterMark\\":null}",\n'
 '  "errors": [],\n'
```

Warnings are common with some source file and skill combinations and don't always indicate a problem. Many warnings are benign. For example, if you index a JPEG file that doesn't have text, you'll see the warning in this screenshot.

```python
#Determine if the indexer is still running
r = requests.get(endpoint + "/indexers/" + indexer_name + "/status", headers=headers,params=params)
pprint(json.dumps(r.json(), indent=1))
```

```
'    "message": "Truncated extracted text to 65536 characters."\n'
'   },\n'
'   {\n'
'    "key": '
'"https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/NYSE_LNKD_2015.PDF",\n'
'    "message": "Truncated extracted text to 65536 characters."\n'
'   },\n'
'   {\n'
'    "key": '
'"https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/create-service-full-portal.png",\n'
'    "message": "Skill #1: Input text is missing. No entities will be '
'returned\\r\\nSkill #2: Input text is missing. No language will be '
'returned\\r\\nSkill #3: Input text is missing\\r\\n"\n'
'   },\n'
'   {\n'
'    "key": '
'"https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/create-search-service.png",\n'
'    "message": "Skill #1: Input text is missing. No entities will be '
'returned\\r\\nSkill #2: Input text is missing. No language will be '
```

5 - Search

After indexing is finished, run queries that return the contents of the index or individual fields.

First, get the index definition showing all of the fields. Visual Studio Code limits the output to 30 lines by default, but provides an option to open the output in a text editor. Use that option to view the full output. The output is the index schema, with the name, type, and attributes of each field.

Python

```
# Query the service for the index definition
r = requests.get(endpoint + "/indexes/" + index_name,
                 headers=headers, params=params)
pprint(json.dumps(r.json(), indent=1))
```

Next, submit a second query for `"*"` to return all contents of a single field, such as `organizations`. See Search Documents (REST API) for more information about the request.

Python

```
# Query the index to return the contents of organizations
r = requests.get(endpoint + "/indexes/" + index_name +
                 "/docs?&search=*&$select=organizations", headers=headers,
params=params)
pprint(json.dumps(r.json(), indent=1))
```

If you'd like to continue testing from this notebook, repeat the above commands using other fields: `content`, `languageCode`, `keyPhrases`, and `organizations` in this exercise.

> **Tip**
>
> A better search experience might be switching to **Search Explorer** in the Azure portal or **creating a demo search app** from the index you just created.

Reset and rerun

In the early stages of development, it's practical to delete the objects from Azure Cognitive Search and allow your code to rebuild them. Resource names are unique. Deleting an object lets you recreate it using the same name.

You can use the portal to delete indexes, indexers, data sources, and skillsets. When you delete the indexer, you can optionally, selectively delete the index, skillset, and data source at the same time.

You can also delete them using a script. The following script deletes a skillset.

Python

```python
# delete the skillset
r = requests.delete(endpoint + "/skillsets/" + skillset_name,
                    headers=headers, params=params)
pprint(json.dumps(r.json(), indent=1))
```

Status code 204 is returned on successful deletion.

Takeaways

This tutorial demonstrates the basic steps for building an enriched indexing pipeline through the creation of component parts: a data source, skillset, index, and indexer.

Built-in skills were introduced, along with skillset definitions and a way to chain skills together through inputs and outputs. You also learned that `outputFieldMappings` in the indexer definition is required for routing enriched values from the pipeline into a searchable index on an Azure Cognitive Search service.

Finally, you learned how to test the results and reset the system for further iterations. You learned that issuing queries against the index returns the output created by the enriched indexing pipeline. In this release, there's a mechanism for viewing internal

constructs (enriched documents created by the system). You also learned how to check the indexer status and what objects must be deleted before rerunning a pipeline.

Clean up resources

When you're working in your own subscription, at the end of a project, it's a good idea to remove the resources that you no longer need. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with all of the objects in an AI enrichment pipeline, let's take a closer look at skillset definitions and individual skills.

How to create a skillset

Tutorial: Debug a skillset using Debug Sessions

Article • 03/24/2023 • 11 minutes to read

Skillsets coordinate a series of actions that analyze or transform content, where the output of one skill becomes the input of another. When inputs depend on outputs, mistakes in skillset definitions and field associations can result in missed operations and data.

Debug sessions in the Azure portal provides a holistic visualization of a skillset. Using this tool, you can drill down to specific steps to easily see where an action might be falling down.

In this article, you'll use **Debug sessions** to find and fix missing inputs and outputs. The tutorial is all-inclusive. It provides sample data, a Postman collection that creates objects, and instructions for debugging problems in the skillset.

Prerequisites

Before you begin, have the following prerequisites in place:

- An active subscription. Create an account for free.
- Azure Cognitive Search. Create a service or find an existing service under your current subscription. You can use a free service for this tutorial.
- Azure Storage account with Blob storage, used for hosting sample data, and for persisting cached data created during a debug session.
- Postman app and a Postman collection to create objects using the REST APIs.
- Sample PDFs (clinical trials).

> **Note**
>
> This tutorial also uses **Azure Cognitive Services** for language detection, entity recognition, and key phrase extraction. Because the workload is so small, Cognitive Services is tapped behind the scenes for free processing for up to 20 transactions. This means that you can complete this exercise without having to create a billable Cognitive Services resource.

Set up your data

This section creates the sample data set in Azure Blob Storage so that the indexer and skillset have content to work with.

1. Download sample data (clinical-trials-pdf-19), consisting of 19 files.
2. Create an Azure storage account or find an existing account.
   - Choose the same region as Azure Cognitive Search to avoid bandwidth charges.
   - Choose the StorageV2 (general purpose V2) account type.
3. Navigate to the Azure Storage services pages in the portal and create a Blob container. Best practice is to specify the access level "private". Name your container `clinicaltrialdataset`.
4. In container, click **Upload** to upload the sample files you downloaded and unzipped in the first step.
5. While in the portal, get and save off the connection string for Azure Storage. You'll need it for the REST API calls that index data. You can get the connection string from **Settings** > **Access Keys** in the portal.

Get a key and URL

REST calls require the service URL and an access key on every request. A search service is created with both, so if you added Azure Cognitive Search to your subscription, follow these steps to get the necessary information:

1. Sign in to the Azure portal, and in your search service **Overview** page, get the URL. An example endpoint might look like `https://mydemo.search.windows.net`.
2. In **Settings** > **Keys**, get an admin key for full rights on the service. There are two interchangeable admin keys, provided for business continuity in case you need to roll one over. You can use either the primary or secondary key on requests for adding, modifying, and deleting objects.

All requests require an api-key on every request sent to your service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Create data source, skillset, index, and indexer

In this section, Postman and a provided collection are used to create the Cognitive Search data source, skillset, index, and indexer. If you're unfamiliar with Postman, see this quickstart.

You will need the Postman collection created for this tutorial to complete this task.

1. Start Postman and import the "DebugSessions.postman_collection.json" collection. Under **Files** > **New**, select the collection.

2. After the collection is imported, expand the actions list (...).

3. Select **Edit** to set variables used in each request, and then **Save**.

Current value	Description
searchService	The name of your search service (for example, if the endpoint is `https://mydemo.search.windows.net`, then the service name is "mydemo".
apiKey	The primary or secondary key obtained from the **Keys** page of your search service.

Current value	Description
storageConnectionString	The connection string obtained from the **Access Keys** page of your Azure Storage account.
containerName	The name of the container you created for the sample data.

4. Verify that the collection you imported contains four REST calls, used to create objects in this tutorial.

   - CreateDataSource adds `clinical-trials-ds`
   - CreateSkillset adds `clinical-trials-ss`
   - CreateIndex adds `clinical-trials`
   - CreateIndexer adds `clinical-trials-idxr`

5. Open each request in turn, and select **Send** to send each request to the search service. The last one will take several minutes to complete.

6. Close Postman and return to the Azure portal.

Check results in the portal

The sample code intentionally creates a buggy index as a consequence of problems that occurred during skillset execution. The problem is that the index is missing data.

1. In Azure portal, on the search service **Overview** page, select the **Indexes** tab.

2. Select *clinical-trials*.

3. Enter this query string: `$select=metadata_storage_path, organizations, locations&$count=true` to return fields for specific documents (identified by the unique `metadata_storage_path` field).

4. Select **Search** to run the query. You should see empty values for "organizations" and "locations".

These fields should have been populated through the skillset's Entity Recognition skill, used to detect organizations and locations anywhere within the blob's content. In the next exercise, you'll debug the skillset to determine what went wrong.

Another way to investigate errors and warnings is through the Azure portal.

1. Open the **Indexers** tab and select *clinical-trials-idxr*.

   Notice that while the indexer job succeeded overall, there were warnings.

2. Select **Success** to view the warnings (if there were mostly errors, the detail link would be **Failed**). You'll see a long list of every warning emitted by the indexer.

Status	Last run	Duration	Docs succeeded	Errors/Warnings
Success	1/1/2022, 11:42:52	20.01 s	19/19	0/59

Start your debug session

1. From the search service **Overview** page, click the **Debug sessions** tab.

2. Select + **New Debug Session**.

3. Give the session a name.

4. Connect the session to your storage account.

5. In Indexer template, provide the indexer name. The indexer has references to the data source, the skillset, and index.

6. Accept the default document choice for the first document in the collection. A debug session only works with a single document. You can choose which document to debug, or just use the first one.

7. **Save** the session. Saving the session will kick off the enrichment pipeline as defined by the skillset for the selected document.

8. When the debug session has finished initializing, the session defaults to the **AI Enrichments** tab, highlighting the **Skill Graph**. The Skill Graph provides a visual hierarchy of the skillset and its order of execution sequentially and in parallel.

Find issues with the skillset

Any issues reported by the indexer can be found in the adjacent **Errors/Warnings** tab.

Notice that the **Errors/Warnings** tab will provide a much smaller list than the one displayed earlier because this list is only detailing the errors for a single document. Like the list displayed by the indexer, you can click on a warning message and see the details of this warning.

Select **Errors/Warnings** to review the notifications. You should see four:

- "Could not execute skill because one or more skill input was invalid. Required skill input is missing. Name: 'text', Source: '/document/content'."
- "Could not map output field 'locations' to search index. Check the 'outputFieldMappings' property of your indexer. Missing value '/document/merged_content/locations'."
- "Could not map output field 'organizations' to search index. Check the 'outputFieldMappings' property of your indexer. Missing value '/document/merged_content/organizations'."
- "Skill executed but may have unexpected results because one or more skill input was invalid. Optional skill input is missing. Name: 'languageCode', Source: '/document/languageCode'. Expression language parsing issues: Missing value '/document/languageCode'."

Many skills have a "languageCode" parameter. By inspecting the operation, you can see that this language code input is missing from the `EntityRecognitionSkillV3.#1`, which is the same Entity Recognition skill that is having trouble with 'locations' and 'organizations' output.

Because all four notifications are about this skill, your next step is to debug this skill. If possible, start by solving input issues first before moving on to output issues.

Fix missing skill input values

In the **Errors/Warnings** tab, there are two missing inputs for an operation labeled `EntityRecognitionSkillV3.#1`. The detail of the first error explains that a required input for 'text' is missing. The second indicates a problem with an input value "/document/languageCode".

1. In **AI Enrichments** > **Skill Graph**, select the skill labeled **#1** to display its details in the right pane.
2. Select the **Executions** tab and locate the input for "text".
3. Select the **</>** symbol to pop open the Expression Evaluator. The displayed result for this input doesn’t look like a text input. It looks like a series of new line characters `\n \n\n\n\n` instead of text. The lack of text means that no entities can be identified, so either this document fails to meet the prerequisites of the skill, or there is another input that should be used instead.

4. Switch the left pane to **Enriched Data Structure** and scroll down the list of enrichment nodes for this document. Notice the `\n \n\n\n\n` for "content" has no originating source, but another value for "merged_content" has OCR output. Although there is no indication, the content of this PDF appears to be a JPEG file, as evidenced by the extracted and processed text in "merged_content".

5. In the right pane, select **Executions** for the #1 skill and open the Expression Evaluator </> for the input "text".

6. Change the expression from `/document/content` to `/document/merged_content`, and then select **Evaluate**. Notice that the content is now a chunk of text, and thus actionable for entity recognition.

7. Switch to **Skill JSON Editor**.

8. Change `/document/content` to `/document/merged_content`.

9. Select **Save** in the Skill Details pane.

10. Select **Run** in the session's window menu. This will kick off another execution of the skillset using the document.

11. Once the debug session execution completes, check the Errors/Warnings tab and it will show that the error for text input is gone, but the other warnings remain. The next step is to address the warning about "languageCode".

12. Select the **Executions** tab and locate the input for "languageCode".

13. Select the </> symbol to pop open the Expression Evaluator. Notice the confirmation that the "languageCode" property is not a valid input.

There are two ways to research this error. The first is to look at where the input is coming from - what skill in the hierarchy is supposed to produce this result? The Executions tab in the skill details pane should display the source of the input. If there is no source, this indicates a field mapping error.

1. In the **Executions** tab, check the INPUTS and find "languageCode". There is no source for this input listed.

2. Switch the left pane to **Enriched Data Structure**. Scroll down the list of enrichment nodes for this document. Notice that there is no "languageCode" node, but there is one for "language". So, there is a typo in the skill settings.

3. Still in the **Enriched Data Structure**, open the Expression Evaluator </> for the "language" node and copy the expression `/document/language`.

4. In the right pane, select **Skill Settings** for the #1 skill and open the Expression Evaluator </> for the input "languageCode".

5. Paste the new value, `/document/language` into the Expression box and click **Evaluate**. It should display the correct input "en".

6. Select **Save**.

7. Select **Run**.

After the debug session execution completes, check the Errors/Warnings tab and it will show that all of the input warnings are gone. There now remain just the two warnings about output fields for organizations and locations.

Fix missing skill output values

The messages say to check the 'outputFieldMappings' property of your indexer, so lets start there.

1. Go to **Skill Graph** and select Output Field Mappings. The mappings are actually correct, but normally you would check the index definition to ensure that fields exist for "locations" and "organizations".

2. If there is no problem with the index, the next step is to check skill outputs. As before, select the **Enriched Data Structure**, and scroll the nodes to find "locations" and "organizations". Notice that the parent is "content" instead of "merged_content". The context is wrong.

3. Switch back to **Skill Graph** and select the entity recognition skill.

4. Navigate the **Skill Settings** to find "context."

5. Double-click the setting for "context" and edit it to read '/document/merged_content'.

6. Select **Save**.

7. Select **Run**.

All of the errors have been resolved.

Commit changes to the skillset

When the debug session was initiated, the search service created a copy of the skillset. This was done to protect the original skillset on your search service. Now that you have finished debugging your skillset, the fixes can be committed (overwrite the original skillset).

Alternatively, if you aren't ready to commit changes, you can save the debug session and reopen it later.

1. Select **Commit changes** in the main Debug sessions menu.
2. Select **OK** to confirm that you wish to update your skillset.
3. Close Debug session and select the **Indexers** tab.
4. Open your 'clinical-trials-idxr'.
5. Select **Reset**.
6. Select **Run**. Select **OK** to confirm.

When the indexer has finished running, there should be a green checkmark and the word Success next to the time stamp for the latest run in the **Execution history** tab. To ensure that the changes have been applied:

1. In the search Overview page, select the **Index** tab.
2. Open the 'clinical-trials' index and in the Search explorer tab, enter this query string: `$select=metadata_storage_path, organizations, locations&$count=true` to return fields for specific documents (identified by the unique `metadata_storage_path` field).
3. Select **Search**.

The results should show that organizations and locations are now populated with the expected values.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you are using a free service, remember that you are limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

This tutorial touched on various aspects of skillset definition and processing. To learn more about concepts and workflows, refer to the following articles:

- How to map skillset output fields to fields in a search index
- Skillsets in Azure Cognitive Search
- How to configure caching for incremental enrichment

Tutorial: Create a custom analyzer for phone numbers

Article • 03/24/2023 • 12 minutes to read

Analyzers are a key component in any search solution. To improve the quality of search results, it's important to understand how analyzers work and impact these results.

In some cases, like with a free text field, simply selecting the correct language analyzer will improve search results. However, some scenarios such as accurately searching phone numbers, URLs, or emails may require the use of custom analyzers.

This tutorial uses Postman and Azure Cognitive Search's REST APIs to:

- ✓ Explain how analyzers work
- ✓ Define a custom analyzer for searching phone numbers
- ✓ Test how the custom analyzer tokenizes text
- ✓ Create separate analyzers for indexing and searching to further improve results

Prerequisites

The following services and tools are required for this tutorial.

- Postman app
- Create or find an existing search service

Download files

Source code for this tutorial is in the custom-analyzers folder in the Azure-Samples/azure-search-postman-samples GitHub repository.

1 - Create Azure Cognitive Search service

To complete this tutorial, you'll need an Azure Cognitive Search service, which you can create in the portal. You can use the Free tier to complete this walkthrough.

For the next step, you'll need to know the name of your search service and its API Key. If you're unsure how to find those items, check out this REST quickstart.

2 - Set up Postman

Next, start Postman and import the collection you downloaded from Azure-Samples/azure-search-postman-samples.

To import the collection, go to **Files** > **Import**, then select the collection file you'd like to import.

For each request, you need to:

1. Replace `<YOUR-SEARCH-SERVICE>` with the name of your search service.
2. Replace `<YOUR-ADMIN-API-KEY>` with either the primary or secondary key of your search service.

If you're unfamiliar with Postman, see Explore Azure Cognitive Search REST APIs.

3 - Create an initial index

In this step, we'll create an initial index, load documents into the index, and then query the documents to see how our initial searches perform.

Create index

We'll start by creating a simple index called `tutorial-basic-index` with two fields: `id` and `phone_number`. We haven't defined an analyzer yet so the `standard.lucene` analyzer will be used by default.

To create the index, we send the following request:

HTTP

```
PUT https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-basic-index?api-version=2019-05-06
  Content-Type: application/json
  api-key: <YOUR-ADMIN-API-KEY>

  {
    "fields": [
```

```
      {
        "name": "id",
        "type": "Edm.String",
        "key": true,
        "searchable": true,
        "filterable": false,
        "facetable": false,
        "sortable": true
      },
      {
        "name": "phone_number",
        "type": "Edm.String",
        "sortable": false,
        "searchable": true,
        "filterable": false,
        "facetable": false
      }
    ]
  }
```

Load data

Next, we'll load data into the index. In some cases, you may not have control over the format of the phone numbers ingested so we'll test against different kinds of formats. Ideally, a search solution will return all matching phone numbers regardless of their format.

Data is loaded into the index using the following request:

HTTP

```
POST https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-basic-index/docs/index?api-version=2019-05-06
  Content-Type: application/json
  api-key: <YOUR-ADMIN-API-KEY>

  {
    "value": [
      {
        "@search.action": "upload",
        "id": "1",
        "phone_number": "425-555-0100"
      },
      {
        "@search.action": "upload",
        "id": "2",
        "phone_number": "(321) 555-0199"
      },
      {
        "@search.action": "upload",
        "id": "3",
```

```
        "phone_number": "+1 425-555-0100"
      },
      {
        "@search.action": "upload",
        "id": "4",
        "phone_number": "+1 (321) 555-0199"
      },
      {
        "@search.action": "upload",
        "id": "5",
        "phone_number": "4255550100"
      },
      {
        "@search.action": "upload",
        "id": "6",
        "phone_number": "13215550199"
      },
      {
        "@search.action": "upload",
        "id": "7",
        "phone_number": "425 555 0100"
      },
      {
        "@search.action": "upload",
        "id": "8",
        "phone_number": "321.555.0199"
      }
    ]
  }
```

With the data in the index, we're ready to start searching.

Search

To make the search intuitive, it's best to not expect users to format queries in a specific way. A user could search for `(425) 555-0100` in any of the formats we showed above and will still expect results to be returned. In this step, we'll test out a couple of sample queries to see how they perform.

We start by searching `(425) 555-0100`:

HTTP

```
GET https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-basic-index/docs?api-version=2019-05-06&search=(425) 555-0100
  Content-Type: application/json
  api-key: <YOUR-ADMIN-API-KEY>
```

This query returns **three out of four expected results** but also returns **two unexpected results**:

JSON

```
{
    "value": [
        {
            "@search.score": 0.05634898,
            "phone_number": "+1 425-555-0100"
        },
        {
            "@search.score": 0.05634898,
            "phone_number": "425 555 0100"
        },
        {
            "@search.score": 0.05634898,
            "phone_number": "425-555-0100"
        },
        {
            "@search.score": 0.020766128,
            "phone_number": "(321) 555-0199"
        },
        {
            "@search.score": 0.020766128,
            "phone_number": "+1 (321) 555-0199"
        }
    ]
}
```

Next, let's search a number without any formatting `4255550100`

HTTP

```
GET https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-basic-index/docs?api-version=2019-05-06&search=4255550100
  api-key: <YOUR-ADMIN-API-KEY>
```

This query does even worse, only returning **one of four correct matches**.

JSON

```
{
    "value": [
        {
            "@search.score": 0.6015292,
            "phone_number": "4255550100"
        }
    ]
}
```

If you find these results confusing, you're not alone. In the next section, we'll dig into why we're getting these results.

4 - Debug search results

To understand these search results, it's important to first understand how analyzers work. From there, we can test the default analyzer using the Analyze Text API and then create an analyzer that meets our needs.

How analyzers work

An analyzer is a component of the full text search engine responsible for processing text in query strings and indexed documents. Different analyzers manipulate text in different ways depending on the scenario. For this scenario, we need to build an analyzer tailored to phone numbers.

Analyzers consist of three components:

- **Character filters** that remove or replace individual characters from the input text.
- A **Tokenizer** that breaks the input text into tokens, which become keys in the search index.
- **Token filters** that manipulate the tokens produced by the tokenizer.

In the diagram below, you can see how these three components work together to tokenize a sentence:

These tokens are then stored in an inverted index, which allows for fast, full-text searches. An inverted index enables full-text search by mapping all unique terms extracted during lexical analysis to the documents in which they occur. You can see an example in the diagram below:

All of search comes down to searching for the terms stored in the inverted index. When a user issues a query:

1. The query is parsed and the query terms are analyzed.
2. The inverted index is then scanned for documents with matching terms.
3. Finally, the retrieved documents are ranked by the scoring algorithm.

If the query terms don't match the terms in your inverted index, results won't be returned. To learn more about how queries work, see this article on full text search.

> ⓘ **Note**
>
> **Partial term queries** are an important exception to this rule. These queries (prefix query, wildcard query, regex query) bypass the lexical analysis process unlike regular term queries. Partial terms are only lowercased before being matched against terms in the index. If an analyzer isn't configured to support these types of queries, you'll often receive unexpected results because matching terms don't exist in the index.

Test analyzer using the Analyze Text API

Azure Cognitive Search provides an Analyze Text API that allows you to test analyzers to understand how they process text.

The Analyze Text API is called using the following request:

HTTP

```
POST https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-basic-index/analyze?api-version=2019-05-06
  Content-Type: application/json
  api-key: <YOUR-ADMIN-API-KEY>

  {
    "text": "(425) 555-0100",
    "analyzer": "standard.lucene"
  }
```

The API then returns a list of the tokens extracted from the text. You can see that the standard Lucene analyzer splits the phone number into three separate tokens:

JSON

```
{
    "tokens": [
        {
            "token": "425",
            "startOffset": 1,
            "endOffset": 4,
            "position": 0
        },
        {
            "token": "555",
            "startOffset": 6,
            "endOffset": 9,
            "position": 1
        },
        {
            "token": "0100",
            "startOffset": 10,
            "endOffset": 14,
            "position": 2
        }
    ]
}
```

Conversely, the phone number `4255550100` formatted without any punctuation is tokenized into a single token.

JSON

```
{
  "text": "4255550100",
```

```
  "analyzer": "standard.lucene"
}
```

JSON

```
{
    "tokens": [
        {
            "token": "4255550100",
            "startOffset": 0,
            "endOffset": 10,
            "position": 0
        }
    ]
}
```

Keep in mind that both query terms and the indexed documents are analyzed. Thinking back to the search results from the previous step, we can start to see why those results were returned.

In the first query, the incorrect phone numbers were returned because one of their terms, `555`, matched one of the terms we searched. In the second query, only the one number was returned because it was the only record that had a term matching `4255550100`.

5 - Build a custom analyzer

Now that we understand the results we're seeing, let's build a custom analyzer to improve the tokenization logic.

The goal is to provide intuitive search against phone numbers no matter what format the query or indexed string is in. To achieve this result, we'll specify a character filter, a tokenizer, and a token filter.

Character filters

Character filters are used to process text before it's fed into the tokenizer. Common uses of character filters include filtering out HTML elements or replacing special characters.

For phone numbers, we want to remove whitespace and special characters because not all phone number formats contain the same special characters and spaces.

JSON

```
"charFilters": [
    {
      "@odata.type": "#Microsoft.Azure.Search.MappingCharFilter",
      "name": "phone_char_mapping",
      "mappings": [
        "-=>",
        "(=>",
        ")=>",
        "+=>",
        ".=>",
        "\\u0020=>"
      ]
    }
  ]
```

The filter above will remove `-` `(` `)` `+` `.` and spaces from the input.

Input	Output
`(321) 555-0199`	`3215550199`
`321.555.0199`	`3215550199`

Tokenizers

Tokenizers split text into tokens and discard some characters, such as punctuation, along the way. In many cases, the goal of tokenization is to split a sentence into individual words.

For this scenario, we'll use a keyword tokenizer, `keyword_v2`, because we want to capture the phone number as a single term. Note that this isn't the only way to solve this problem. See the Alternate approaches section below.

Keyword tokenizers always output the same text it was given as a single term.

Input	Output
`The dog swims.`	`[The dog swims.]`
`3215550199`	`[3215550199]`

Token filters

Token filters will filter out or modify the tokens generated by the tokenizer. One common use of a token filter is to lowercase all characters using a lowercase token filter.

Another common use is filtering out stopwords such as `the`, `and`, or `is`.

While we don't need to use either of those filters for this scenario, we'll use an nGram token filter to allow for partial searches of phone numbers.

JSON

```
"tokenFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.NGramTokenFilterV2",
    "name": "custom_ngram_filter",
    "minGram": 3,
    "maxGram": 20
  }
]
```

NGramTokenFilterV2

The nGram_v2 token filter splits tokens into n-grams of a given size based on the `minGram` and `maxGram` parameters.

For the phone analyzer, we set `minGram` to `3` because that is the shortest substring we expect users to search. `maxGram` is set to `20` to ensure that all phone numbers, even with extensions, will fit into a single n-gram.

The unfortunate side effect of n-grams is that some false positives will be returned. We'll fix this in step 7 by building out a separate analyzer for searches that doesn't include the n-gram token filter.

Input	Output
`[12345]`	`[123, 1234, 12345, 234, 2345, 345]`
`[3215550199]`	`[321, 3215, 32155, 321555, 3215550, 32155501, 321555019, 3215550199, 215, 2155, 21555, 215550, ... ]`

Analyzer

With our character filters, tokenizer, and token filters in place, we're ready to define our analyzer.

JSON

```
"analyzers": [
  {
```

```
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "phone_analyzer",
    "tokenizer": "custom_tokenizer_phone",
    "tokenFilters": [
      "custom_ngram_filter"
    ],
    "charFilters": [
      "phone_char_mapping"
    ]
  }
]
```

Input	Output
`12345`	`[123, 1234, 12345, 234, 2345, 345]`
`(321) 555-0199`	`[321, 3215, 32155, 321555, 3215550, 32155501, 321555019, 3215550199, 215, 2155, 21555, 215550, ... ]`

Notice that any of the tokens in the output can now be searched. If our query includes any of those tokens, the phone number will be returned.

With the custom analyzer defined, recreate the index so that the custom analyzer will be available for testing in the next step. For simplicity, the Postman collection creates a new index named `tutorial-first-analyzer` with the analyzer we defined.

6 - Test the custom analyzer

After creating the index, you can now test out the analyzer we created using the following request:

HTTP

```
POST https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/tutorial-first-analyzer/analyze?api-version=2019-05-06
  Content-Type: application/json
  api-key: <YOUR-ADMIN-API-KEY>

  {
    "text": "+1 (321) 555-0199",
    "analyzer": "phone_analyzer"
  }
```

You will then be able to see the collection of tokens resulting from the phone number:

JSON

```
{
    "tokens": [
        {
            "token": "132",
            "startOffset": 1,
            "endOffset": 17,
            "position": 0
        },
        {
            "token": "1321",
            "startOffset": 1,
            "endOffset": 17,
            "position": 0
        },
        {
            "token": "13215",
            "startOffset": 1,
            "endOffset": 17,
            "position": 0
        },
        ...
        ...
        ...
    ]
}
```

7 - Build a custom analyzer for queries

After making some sample queries against the index with the custom analyzer, you'll find that recall has improved and all matching phone numbers are now returned. However, the n-gram token filter causes some false positives to be returned as well. This is a common side effect of an n-gram token filter.

To prevent false positives, we'll create a separate analyzer for querying. This analyzer will be the same as the analyzer we created already but **without** the `custom_ngram_filter`.

JSON

```
    {
      "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
      "name": "phone_analyzer_search",
      "tokenizer": "custom_tokenizer_phone",
      "tokenFilters": [],
      "charFilters": [
        "phone_char_mapping"
      ]
    }
```

In the index definition, we then specify both an `indexAnalyzer` and a `searchAnalyzer`.

JSON

```
    {
      "name": "phone_number",
      "type": "Edm.String",
      "sortable": false,
      "searchable": true,
      "filterable": false,
      "facetable": false,
      "indexAnalyzer": "phone_analyzer",
      "searchAnalyzer": "phone_analyzer_search"
    }
```

With this change, you're all set. Recreate the index, index the data, and test the queries again to verify the search works as expected. If you're using the Postman collection, it will create a third index named `tutorial-second-analyzer`.

Alternate approaches

The analyzer above was designed to maximize the flexibility for search. However, it does so at the cost of storing many potentially unimportant terms in the index.

The example below shows a different analyzer that can also be used for this task.

The analyzer works well except for input data such as `14255550100` that makes it difficult to logically chunk the phone number. For example, the analyzer wouldn't be able to separate the country code, `1`, from the area code, `425`. This discrepancy would lead to the number above not being returned if a user didn't include a country code in their search.

JSON

```
"analyzers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.CustomAnalyzer",
    "name": "phone_analyzer_shingles",
    "tokenizer": "custom_tokenizer_phone",
    "tokenFilters": [
      "custom_shingle_filter"
    ]
  }
],
"tokenizers": [
  {
    "@odata.type": "#Microsoft.Azure.Search.StandardTokenizerV2",
```

```
    "name": "custom_tokenizer_phone",
    "maxTokenLength": 4
  }
],
"tokenFilters": [
  {
    "@odata.type": "#Microsoft.Azure.Search.ShingleTokenFilter",
    "name": "custom_shingle_filter",
    "minShingleSize": 2,
    "maxShingleSize": 6,
    "tokenSeparator": ""
  }
]
```

You can see in the example below that the phone number is split into the chunks you would normally expect a user to be searching for.

Input	Output
`(321) 555-0199`	`[321, 555, 0199, 321555, 5550199, 3215550199]`

Depending on your requirements, this may be a more efficient approach to the problem.

Reset and rerun

For simplicity, this tutorial had you create three new indexes. However, it's common to delete and recreate indexes during the early stages of development. You can delete an index in the Azure portal or using the following API call:

HTTP

```
DELETE https://<YOUR-SEARCH-SERVICE-
NAME>.search.windows.net/indexes/tutorial-basic-index?api-version=2019-05-06
  api-key: <YOUR-ADMIN-API-KEY>
```

Takeaways

This tutorial demonstrated the process for building and testing a custom analyzer. You created an index, indexed data, and then queried against the index to see what search results were returned. From there, you used the Analyze Text API to see the lexical analysis process in action.

While the analyzer defined in this tutorial offers an easy solution for searching against phone numbers, this same process can be used to build a custom phone analyzer for

any scenario you may have.

Clean up resources

When you're working in your own subscription, it's a good idea to remove the resources that you no longer need at the end of a project. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the All resources or Resource groups link in the left-navigation pane.

Next steps

Now that you're familiar with how to create a custom analyzer, let's take a look at all of the different filters, tokenizers, and analyzers available to you to build a rich search experience.

Custom Analyzers in Azure Cognitive Search

Tutorial: Query a Cognitive Search index from Power Apps

Article • 03/08/2023 • 8 minutes to read

Use the rapid application development environment of Power Apps to create a custom app for your searchable content in Azure Cognitive Search.

In this tutorial, you learn how to:

- ✓ Connect to Azure Cognitive Search
- ✓ Set up a query request
- ✓ Visualize results in a canvas app

If you don't have an Azure subscription, open a free account before you begin.

Prerequisites

- Power Apps account
- Hotels-sample index hosted on your search service
- Query API key

1 - Create a custom connector

A connector in Power Apps is a data source connection. In this step, create a custom connector to connect to a search index in the cloud.

1. Sign in to Power Apps.
2. On the left, expand **... More**. Find, pin, and then select **Custom Connectors**.

3. Select + **New custom connector**, and then select **Create from blank**.

4. Give your custom connector a name (for example, *AzureSearchQuery*), and then select **Continue**.

5. Enter information in the General Page:

   - Icon background color (for instance, #007ee5)
   - Description (for instance, "A connector to Azure Cognitive Search")
   - In the Host, enter your search service URL (such as `<yourservicename>.search.windows.net`)
   - For Base URL, enter "/"

6. In the Security Page, set *API Key* as the **Authentication Type**, set both the parameter label and parameter name to *api-key*. For **Parameter location**, select *Header* as shown in the following screenshot.

7. In the Definitions Page, select + **New Action** to create an action that queries the index. Enter the value "Query" for the summary and the name of the operation ID. Enter a description like "*Queries the search index*".

8. Scroll down. In Requests, select + **Import from sample** button to configure a query request to your search service:

   - Select the verb `GET`

- For the URL, enter a sample query for your search index (`search=*` returns all documents, `$select=` lets you choose fields). The API version is required. Fully specified, a URL might look like this: `https://mydemo.search.windows.net/indexes/hotels-sample-index/docs?search=*&$select=HotelName,Description,Address/City&api-version=2020-06-30`

- For Headers, type `Content-Type`. You'll set the value to `application/json` in a later step.

  Power Apps uses the syntax in the URL to extract parameters from the query: search, select, and api-version parameters become configurable as you progress through the wizard.

9. Select **Import** to auto-fill the Request. Complete setting the parameter metadata by clicking the **...** symbol next to each of the parameters. Select **Back** to return to the Request page after each parameter update.

10. For *search*: Set `*` as the **default value**, set **required** as *False* and set **visibility** to *none*.

11. For *select*: Set `HotelName,Description,Address/City` as the **default value**, set **required** to *False*, and set **visibility** to *none*.

12. For *api-version*: Set `2020-06-30` as the **default value**, set **required** to *True*, and set **visibility** as *internal*.

13. For *Content-Type*: Set to `application/json`.

14. After making these changes, toggle to the **Swagger Editor** view. In the parameters section you should see the following configuration:

```JSON
parameters:
  - {name: search, in: query, required: false, type: string, default:
'*'}
  - {name: $select, in: query, required: false, type: string, default:
'HotelName,Description,Address/City'}
  - {name: api-version, in: query, required: true, type: string,
default: '2020-06-30',
    x-ms-visibility: internal}
  - {name: Content-Type, in: header, required: false, type: string}
```

15. Switch back to the wizard and return to the **3. Request** step. Scroll down to the Response section. Select "**Add default response**". This step is critical because it helps Power Apps understand the schema of the response.

16. Paste a sample response. An easy way to capture a sample response is through Search Explorer in the Azure portal. In Search Explorer, you should enter the same

query as you did for the request, but add **$top=2** to constrain results to just two documents: `search=*&$select=HotelName,Description,Address/City&$top=2`.

Power Apps only needs a few results to detect the schema. You can copy the following response into the wizard now, assuming you're using the hotels-sample-index.

JSON

```
{
    "@odata.context":
"https://mydemo.search.windows.net/indexes('hotels-sample-
index')/$metadata#docs(*)",
    "value": [
        {
            "@search.score": 1,
            "HotelName": "Arcadia Resort & Restaurant",
            "Description": "The largest year-round resort in the area
offering more of everything for your vacation – at the best value!
What can you enjoy while at the resort, aside from the mile-long sandy
beaches of the lake? Check out our activities sure to excite both young
and young-at-heart guests. We have it all, including being named
“Property of the Year” and a “Top Ten Resort” by top publications.",
            "Address": {
                "City": "Seattle"
            }
        },
        {
            "@search.score": 1,
            "HotelName": "Travel Resort",
            "Description": "The Best Gaming Resort in the area.  With
elegant rooms & suites, pool, cabanas, spa, brewery & world-class
gaming.  This is the best place to play, stay & dine.",
            "Address": {
                "City": "Albuquerque"
            }
        }
    ]
}
```

Tip

There is a character limit to the JSON response you can enter, so you may want to simplify the JSON before pasting it. The schema and format of the response is more important than the values themselves. For example, the Description field could be simplified to just the first sentence.

17. Select **Import** to add the default response.

18. Select **Create connector** on the top right to save the definition.

19. Select **Close** to close the connector.

2 - Test the connection

When the connector is first created, you need to reopen it from the Custom Connectors list in order to test it. Later, if you make more updates, you can test from within the wizard.

You'll need a query API key for this task. Each time a connection is created, whether for a test run or inclusion in an app, the connector needs the query API key used for connecting to Azure Cognitive Search.

1. On the far left, select **Custom Connectors**.

2. Find your connector in the list (in this tutorial, is "AzureSearchQuery").

3. Select the connector, expand the actions list, and select **View Properties**.

4. Select **Edit** on the top right.

5. Select **5. Test** to open the test page.

6. In Test Operation, select **+ New Connection**.

7. Enter a query API key. This is an Azure Cognitive Search query for read-only access to an index. You can find the key in the Azure portal.

8. In Operations, select the **Test operation** button. If you're successful you should see a 200 status, and in the body of the response you should see JSON that describes the search results.

If the test fails, recheck the inputs. In particular, revisit the sample response and make sure it was created properly. The connector definition should show the expected items for the response.

3 - Visualize results

In this step, create a Power App with a search box, a search button, and a display area for the results. The Power App will connect to the recently created custom connector to get the data from Azure Search.

1. On the left, expand **Apps** > **+ New app** > **Canvas**.

2. Select the type of application. For this tutorial, create a **Blank App** with the **Phone Layout**. Give the app a name, such as "Hotel Finder". Select **Create**. The **Power Apps Studio** appears.

3. In the studio, select the **Data Sources** tab, select **+ Add data**, and then find the new Connector you have just created. In this tutorial, it's called *AzureSearchQuery*. Select **Add a connection**.

   Enter the query API key.

   Now *AzureSearchQuery* is a data source that is available to be used from your application.

4. On the **Insert tab**, add a few controls to the canvas.

5. Insert the following elements:

   - A Text Label with the value "Query:"
   - A Text Input element (call it *txtQuery*, default value: "*")
   - A button with the text "Search"

- A Vertical Gallery called (call it *galleryResults*)

The canvas should look something like this:

6. To make the **Search button** issue a query, paste the following action into **OnSelect**:

```
If(!IsBlank(txtQuery.Text),
    ClearCollect(azResult, AzureSearchQuery.Query({search:
txtQuery.Text}).value))
```

The following screenshot shows the formula bar for the **OnSelect** action.

This action causes the button to update a new collection called *azResult* with the result of the search query, using the text in the *txtQuery* text box as the query term.

> **Note**
>
> Try this if you get a formula syntax error "The function 'ClearCollect' has some invalid functions":
>
> - First, make sure the connector reference is correct. Clear the connector name and begin typing the name of your connector. Intellisense should suggest the right connector and verb.
> - If the error persists, delete and recreate the connector. If there are multiple instances of a connector, the app might be using the wrong one.

7. Link the Vertical Gallery control to the *azResult* collection that was created when you completed the previous step.

   Select the gallery control, and perform the following actions in the properties pane.

   - Set **DataSource** to *azResult*.
   - Select a **Layout** that works for you based on the type of data in your index. In this case, we used the *Title, subtitle and body* layout.
   - **Edit Fields**, and select the fields you would like to visualize.

   Since we provided a sample result when we defined the connector, the app is aware of the fields available in your index.

8. Press **F5** to preview the app.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you're using a free service, remember that you're limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

Next steps

Power Apps enables the rapid application development of custom apps. Now that you know how to connect to a search index, learn more about creating a rich visualize experience in a custom Power App.

Power Apps Learning Catalog

Additional resources

Documentation

Content metadata properties - Azure Cognitive Search

Metadata properties can provide content to fields in a search index. This article lists metadata properties supported in Azure Cognitive Search.

Quickstart: Create a knowledge store in the Azure portal - Azure Cognitive Search

Use the Import data wizard to create a knowledge store used for persisting enriched content. Connect to a knowledge store for analysis from other apps, or send enriched content to downstream processes.

REST Tutorial: AI on Azure blobs - Azure Cognitive Search

Step through an example of text extraction and natural language processing over content in Blob Storage using Postman and the Azure Cognitive Search REST APIs.

Semantic search - Azure Cognitive Search

Learn how Cognitive Search is using deep learning semantic search models from Bing to make search results more intuitive.

Configure semantic search - Azure Cognitive Search

Set a semantic query type to attach the deep learning models of semantic search.

Relevance and scoring - Azure Cognitive Search

Explains the concepts of relevance and scoring in Azure Cognitive Search, and what a developer can do to customize the scoring result.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Quickstart: Create a search index in Python - Azure Cognitive Search

Learn how to create a search index, load data, and run queries using Python, Jupyter Notebook, and the Azure.Documents.Search client library for Python.

Show 5 more

Training

Module

Use custom connectors in a Power Apps canvas app - Training

Learn how to use custom connectors in a Power Apps canvas app.

C# samples for Azure Cognitive Search

Article • 03/10/2023 • 3 minutes to read

Learn about the C# code samples that demonstrate the functionality and workflow of an Azure Cognitive Search solution. These samples use the **Azure Cognitive Search client library** for the **Azure SDK for .NET**, which you can explore through the following links.

Target	Link
Package download	www.nuget.org/packages/Azure.Search.Documents/
API reference	azure.search.documents
API test cases	github.com/Azure/azure-sdk-for-net/tree/master/sdk/search/Azure.Search.Documents/tests
Source code	github.com/Azure/azure-sdk-for-net/tree/master/sdk/search/Azure.Search.Documents/src

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in **Azure/azure-sdk-for-net/tree/master/sdk/search/Azure.Search.Documents/samples** on GitHub.

Samples	Description
"Hello world", synchronously	Demonstrates how to create a client, authenticate, and handle errors using synchronous methods.
"Hello world", asynchronously	Demonstrates how to create a client, authenticate, and handle errors using asynchronous methods.
Service-level operations	Demonstrates how to create indexes, indexers, data sources, skillsets, and synonym maps. This sample also shows you how to get service statistics and how to query an index.
Index operations	Demonstrates how to perform an action on existing index, in this case getting a count of documents stored in the index.
FieldBuilderIgnore	Demonstrates a technique for working with unsupported data types.
Indexing documents (push model)	"Push" model indexing, where you send a JSON payload to an index on a service.

Samples	Description
Encryption key sample	Demonstrates using a customer-managed encryption key to add an extra layer of protection over sensitive content.

Doc samples

Code samples from the Cognitive Search team demonstrate features and workflows. All of the following samples are referenced in tutorials, quickstarts, and how-to articles that explain the code in detail. You can find these samples in **Azure-Samples/azure-search-dotnet-samples** and in **Azure-Samples/search-dotnet-getting-started** on GitHub.

> **Tip**
>
> Try the **Samples browser** to search for Microsoft code samples in GitHub, filtered by product, service, and language.

Code sample	Related article	Purpose
quickstart	Quickstart: Create a search index	Covers the basic workflow for creating, loading, and querying a search index in C# using sample data.
search-website	Tutorial: Add search to web apps	Demonstrates an end-to-end search app that includes a rich client plus components for hosting the app and handling search requests.
DotNetHowTo	How to use the .NET client library	Steps through the basic workflow, but in more detail and with discussion of API usage.
DotNetHowToSynonyms	Example: Add synonyms in C#	Synonym lists are used for query expansion, providing matchable terms that are external to an index.
DotNetToIndexers	Tutorial: Index Azure SQL data	Shows how to configure an Azure SQL indexer that has a schedule, field mappings, and parameters.

Code sample	Related article	Purpose
DotNetHowToEncryptionUsingCMK	How to configure customer-managed keys for data encryption	Shows how to create objects that are encrypted with a customer key.
multiple-data-sources	Tutorial: Index from multiple data sources.	Merges content from two data sources into one search index.
Optimize-data-indexing	Tutorial: Optimize indexing with the push API.	Demonstrates optimization techniques for pushing data into a search index.
tutorial-ai-enrichment	Tutorial: AI-generated searchable content from Azure blobs	Shows how to configure an indexer and skillset.
create-mvc-app	Tutorial: Add search to an ASP.NET Core (MVC) app	While most samples are console applications, this MVC sample uses a web page to front the sample Hotels index, demonstrating basic search, pagination, and other server-side behaviors.

Other samples

The following samples are also published by the Cognitive Search team, but aren't referenced in documentation. Associated readme files provide usage instructions.

Samples	Description
Check storage	Invokes an Azure function that checks search service storage on a schedule.
Export an index	C# console app that partitions and export a large index.

Samples	Description
Query multiple services	Issue a single query across multiple search services and combine the results into a single page.
Index Data Lake Gen2 using Azure AD	Source code demonstrating indexer connections and indexing of Azure Data Lake Gen2 files and folders that are secured through Azure AD and role-based access controls.
azure-search-power-skills	Source code for consumable custom skills that you can incorporate in your won solutions.
Knowledge Mining Solution Accelerator	Includes templates, support files, and analytical reports to help you prototype an end-to-end knowledge mining solution.
Covid-19 Search App repository	Source code repository for the Cognitive Search based Covid-19 Search App
JFK	Learn more about the JFK solution.
Search + QnA Maker Accelerator	A solution combining the power of Search and QnA Maker. See the live demo site.

Additional resources

Documentation

Azure Cognitive Search client library for .NET - Azure for .NET Developers

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Data type map for indexers - Azure Cognitive Search

Source and destination field data type map combinations, used when crawling external data sources for data ingestion in an Azure Cognitive Search index.

Azure.Search.Documents Namespace - Azure for .NET Developers

Explore all classes and interfaces of the Azure.Search.Documents namespace.

C# tutorial on ordering results - Azure Cognitive Search

This C# tutorial demonstrates how to order search results. It builds on a previous hotels project, sorting by primary property, secondary property, and includes a scoring profile to add boosting criteria.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

SearchClient Class (Azure.Search.Documents) - Azure for .NET Developers

Azure Cognitive Search client that can be used to query an index and upload, merge, or delete documents.

Show 5 more

Training

Module

Search data outside the Azure platform in Azure Cognitive Search using Azure Data Factory - Training

Use Azure Data Factory to add data that resides inside or outside the Azure platform into your search indexes.

Java samples for Azure Cognitive Search

Article • 01/27/2023 • 2 minutes to read

Learn about the Java code samples that demonstrate the functionality and workflow of an Azure Cognitive Search solution. These samples use the **Azure Cognitive Search client library** for the **Azure SDK for Java**, which you can explore through the following links.

Target	Link
Package download	search.maven.org/artifact/com.azure/azure-search-documents
API reference	com.azure.search.documents
API test cases	github.com/Azure/azure-sdk-for-java/tree/azure-search-documents_11.1.3/sdk/search/azure-search-documents/src/test
Source code	github.com/Azure/azure-sdk-for-java/tree/azure-search-documents_11.1.3/sdk/search/azure-search-documents/src

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in **Azure/azure-sdk-for-java/tree/master/sdk/search/azure-search-documents/src/samples** on GitHub.

Samples	Description
Search index creation	Demonstrates how to create search indexes.
Synonym creation	Demonstrates how to create synonym maps.
Search indexer creation	Demonstrates how to create indexers.
Search indexer data source creation	Demonstrates how to create indexer data sources, required for indexer-based indexing of supported Azure data sources.
Skillset creation	Demonstrates how to create skillsets that are attached indexers, and that perform AI-based enrichment during indexing.
Load documents	Demonstrates how to upload or merge documents into an index in a data import operation.

Samples	Description
Query syntax	Demonstrates how to set up a basic query.

Doc samples

Code samples from the Cognitive Search team are located in **Azure-Samples/azure-search-java-samples** on GitHub.

Samples	Article
quickstart	Source code for Quickstart: Create a search index in Java and REST.

> **Tip**
>
> Try the **Samples browser** to search for Microsoft code samples in GitHub, filtered by product, service, and language.

Additional resources

Documentation

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Full text query and indexing engine architecture (Lucene) - Azure Cognitive Search

Explore Lucene query processing and document retrieval concepts for full text search, as related to Azure Cognitive Search.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Index operations using Azure Cognitive Search REST APIs

Learn which REST APIs are used to create, delete, and update Azure Cognitive Search index definitions.

Create an indexer - Azure Cognitive Search

Set properties on an indexer to determine data origin and destinations. You can set parameters to modify runtime behaviors.

Model complex data types - Azure Cognitive Search

Nested or hierarchical data structures can be modeled in an Azure Cognitive Search index using ComplexType and Collections data types.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Show 5 more

JavaScript samples for Azure Cognitive Search

Article • 01/06/2023 • 2 minutes to read

Learn about the JavaScript code samples that demonstrate the functionality and workflow of an Azure Cognitive Search solution. These samples use the **Azure Cognitive Search client library** for the **Azure SDK for JavaScript**, which you can explore through the following links.

Target	Link
Package download	www.npmjs.com/package/@azure/search-documents
API reference	@azure/search-documents
API test cases	github.com/Azure/azure-sdk-for-js/tree/master/sdk/search/search-documents/test
Source code	github.com/Azure/azure-sdk-for-js/tree/master/sdk/search/search-documents

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in **azure-sdk-for-js/tree/master/sdk/search/search-documents/samples** on GitHub.

JavaScript SDK samples

Samples	Description
indexes	Demonstrates how to create, update, get, list, and delete search indexes. This sample category also includes a service statistic sample.
dataSourceConnections (for indexers)	Demonstrates how to create, update, get, list, and delete indexer data sources, required for indexer-based indexing of supported Azure data sources.
indexers	Demonstrates how to create, update, get, list, reset, and delete indexers.

Samples	Description
skillSet	Demonstrates how to create, update, get, list, and delete skillsets that are attached indexers, and that perform AI-based enrichment during indexing.
synonymMaps	Demonstrates how to create, update, get, list, and delete synonym maps.

TypeScript samples

Samples	Description
indexes	Demonstrates how to create, update, get, list, and delete search indexes. This sample category also includes a service statistic sample.
dataSourceConnections (for indexers)	Demonstrates how to create, update, get, list, and delete indexer data sources, required for indexer-based indexing of supported Azure data sources.
indexers	Demonstrates how to create, update, get, list, reset, and delete indexers.
skillSet	Demonstrates how to create, update, get, list, and delete skillsets that are attached indexers, and that perform AI-based enrichment during indexing.
synonymMaps	Demonstrates how to create, update, get, list, and delete synonym maps.

Doc samples

Code samples from the Cognitive Search team demonstrate features and workflows. Many of these samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in **Azure-Samples/azure-search-javascript-samples** on GitHub.

Samples	Article
quickstart	Source code for Quickstart: Create a search index in JavaScript. Covers the basic workflow for creating, loading, and querying a search index using sample data.
search-website	Source code for Tutorial: Add search to web apps. Demonstrates an end-to-end search app that includes a rich client plus components for hosting the app and handling search requests.

> **Tip**
>
> Try the **Samples browser** to search for Microsoft code samples in GitHub, filtered by product, service, and language.

Other samples

The following samples are also published by the Cognitive Search team, but aren't referenced in documentation. Associated readme files provide usage instructions.

Samples	Description
azure-search-react-template	React template for Azure Cognitive Search (github.com)

Additional resources

Documentation

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Create an indexer - Azure Cognitive Search

Set properties on an indexer to determine data origin and destinations. You can set parameters to modify runtime behaviors.

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

JavaScript tutorial: Search integration highlights - Azure Cognitive Search

Understand the JavaScript SDK Search integration queries used in the Search-enabled website with this cheat sheet.

Create Indexer (Azure Cognitive Search REST API-version=2020-06-30)

Indexers are resources that automate many aspects of data ingestion into an Azure Cognitive Search indexes. You must use a supported Azure data source to use this API.

Index blobs containing multiple documents - Azure Cognitive Search

Crawl Azure blobs for text content using the Azure Cognitive Search Blob indexer, where each blob might yield one or more search index documents.

Show 5 more

Python samples for Azure Cognitive Search

Article • 02/06/2023 • 2 minutes to read

Learn about the Python code samples that demonstrate the functionality and workflow of an Azure Cognitive Search solution. These samples use the **Azure Cognitive Search client library** for the **Azure SDK for Python**, which you can explore through the following links.

Target	Link
Package download	pypi.org/project/azure-search-documents/
API reference	azure-search-documents
API test cases	github.com/Azure/azure-sdk-for-python/tree/master/sdk/search/azure-search-documents/tests
Source code	github.com/Azure/azure-sdk-for-python/tree/master/sdk/search/azure-search-documents

SDK samples

Code samples from the Azure SDK development team demonstrate API usage. You can find these samples in **azure-sdk-for-python/tree/master/sdk/search/azure-search-documents/samples** on GitHub.

Samples	Description
Authenticate	Demonstrates how to configure a client and authenticate to the service.
Index Create-Read-Update-Delete operations	Demonstrates how to create, update, get, list, and delete search indexes.
Indexer Create-Read-Update-Delete operations	Demonstrates how to create, update, get, list, reset, and delete indexers.
Search indexer data sources	Demonstrates how to create, update, get, list, and delete indexer data sources, required for indexer-based indexing of supported Azure data sources.

Samples	Description
Synonyms	Demonstrates how to create, update, get, list, and delete synonym maps.
Load documents	Demonstrates how to upload or merge documents into an index in a data import operation.
Simple query	Demonstrates how to set up a basic query.
Filter query	Demonstrates setting up a filter expression.
Facet query	Demonstrates working with facets.

Doc samples

Code samples from the Cognitive Search team demonstrate features and workflows. Many of these samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in **Azure-Samples/azure-search-python-samples** on GitHub.

Samples	Article
quickstart	Source code for Quickstart: Create a search index in Python. This article covers the basic workflow for creating, loading, and querying a search index using sample data.
search-website	Source code for Tutorial: Add search to web apps. Demonstrates an end-to-end search app that includes a rich client plus components for hosting the app and handling search requests.
tutorial-ai-enrichment	Source code for Tutorial: Use Python and AI to generate searchable content from Azure blobs. This article shows how to create a blob indexer with a cognitive skillset, where the skillset creates and transforms raw content to make it searchable or consumable.

> **Tip**
>
> Try the **Samples browser** to search for Microsoft code samples in GitHub, filtered by product, service, and language.

Additional resources

Documentation

Quickstart: Create a search index in Python - Azure Cognitive Search

Learn how to create a search index, load data, and run queries using Python, Jupyter Notebook, and the Azure.Documents.Search client library for Python.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Python tutorial: Search integration overview - Azure Cognitive Search

Technical overview and setup for adding search to a website with Python and deploying to Azure Static Web App.

Azure Cognitive Search client library for Python

Map enrichments in indexers - Azure Cognitive Search

Export the enriched content created by a skillset by mapping its output fields to fields in a search index.

azure-search-documents package

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Show 5 more

REST samples for Azure Cognitive Search

Article • 03/24/2023 • 2 minutes to read

Learn about the REST API samples that demonstrate the functionality and workflow of an Azure Cognitive Search solution. These samples use the **Search REST APIs**.

REST is the definitive programming interface for Azure Cognitive Search, and all operations that can be invoked programmatically are available first in REST, and then in SDKs. For this reason, most examples in the documentation leverage the REST APIs to demonstrate or explain important concepts.

REST samples are usually developed and tested on Postman, but you can use any client that supports HTTP calls, including the Postman app. This quickstart explains how to formulate the HTTP request from end-to-end.

Doc samples

Code samples from the Cognitive Search team demonstrate features and workflows. Many of these samples are referenced in tutorials, quickstarts, and how-to articles. You can find these samples in **Azure-Samples/azure-search-postman-samples** on GitHub.

Samples	Article
Quickstart	Source code for Quickstart: Create a search index using REST APIs. This article covers the basic workflow for creating, loading, and querying a search index using sample data.
Tutorial	Source code for Tutorial: Use REST and AI to generate searchable content from Azure blobs. This article shows you how to create a skillset that iterates over Azure blobs to extract information and infer structure.
Debug-sessions	Source code for Tutorial: Diagnose, repair, and commit changes to your skillset. This article shows you how to use a skillset debug session in the Azure portal. REST is used to create the objects used during debug.
custom-analyzers	Source code for Tutorial: Create a custom analyzer for phone numbers. This article explains how to use analyzers to preserve patterns and special characters in searchable content.
knowledge-store	Source code for Create a knowledge store using REST and Postman. This article explains the necessary steps for populating a knowledge store used for knowledge mining workflows.

Samples	Article
projections	Source code for Define projections in a knowledge store. This article explains how to specify the physical data structures in a knowledge store.
index-encrypted-blobs	Source code for How to index encrypted blobs using blob indexers and skillsets. This article shows how to index documents in Azure Blob Storage that have been previously encrypted using Azure Key Vault.

> **Tip**
>
> Try the **Samples browser** to search for Microsoft code samples in GitHub, filtered by product, service, and language.

Other samples

The following samples are also published by the Cognitive Search team, but are not referenced in documentation. Associated readme files provide usage instructions.

Samples	Description
Query-examples	Postman collections demonstrating the various query techniques, including fuzzy search, RegEx and wildcard search, autocomplete, and so on.

Full text search in Azure Cognitive Search

Article • 02/02/2023 • 16 minutes to read

This article is for developers who need a deeper understanding of how Apache Lucene full text search works in Azure Cognitive Search. For text queries, Azure Cognitive Search will seamlessly deliver expected results in most scenarios, but occasionally you might get a result that seems "off" somehow. In these situations, having a background in the four stages of Lucene query execution (query parsing, lexical analysis, document matching, scoring) can help you identify specific changes to query parameters or index configuration that will deliver the desired outcome.

> **Note**
>
> Azure Cognitive Search uses **Apache Lucene** for full text search, but Lucene integration is not exhaustive. We selectively expose and extend Lucene functionality to enable the scenarios important to Azure Cognitive Search.

Architecture overview and diagram

Query execution has four stages:

1. Query parsing
2. Lexical analysis
3. Document retrieval
4. Scoring

A full text search query starts with parsing the query text to extract search terms and operators. There are two parsers so that you can choose between speed and complexity. An analysis phase is next, where individual query terms are sometimes broken down and reconstituted into new forms to cast a broader net over what could be considered as a potential match. The search engine then scans the index to find documents with matching terms and scores each match. A result set is then sorted by a relevance score assigned to each individual matching document. Those at the top of the ranked list are returned to the calling application.

The diagram below illustrates the components used to process a search request.

Key components	Functional description
Query parsers	Separate query terms from query operators and create the query structure (a query tree) to be sent to the search engine.
Analyzers	Perform lexical analysis on query terms. This process can involve transforming, removing, or expanding of query terms.
Index	An efficient data structure used to store and organize searchable terms extracted from indexed documents.
Search engine	Retrieves and scores matching documents based on the contents of the inverted index.

Anatomy of a search request

A search request is a complete specification of what should be returned in a result set. In simplest form, it's an empty query with no criteria of any kind. A more realistic example includes parameters, several query terms, perhaps scoped to certain fields, with possibly a filter expression and ordering rules.

The following example is a search request you might send to Azure Cognitive Search using the REST API.

```
POST /indexes/hotels/docs/search?api-version=2020-06-30
{
    "search": "Spacious, air-condition* +\"Ocean view\"",
    "searchFields": "description, title",
```

```
    "searchMode": "any",
    "filter": "price ge 60 and price lt 300",
    "orderby": "geo.distance(location, geography'POINT(-159.476235
22.227659)')",
    "queryType": "full"
}
```

For this request, the search engine does the following operations:

1. Finds documents where the price is at least $60 and less than $300.

2. Executes the query. In this example, the search query consists of phrases and terms: `"Spacious, air-condition* +\"Ocean view\""` (users typically don't enter punctuation, but including it in the example allows us to explain how analyzers handle it).

   For this query, the search engine scans the description and title fields specified in "searchFields" for documents that contain `"Ocean view"`, and additionally on the term `"spacious"`, or on terms that start with the prefix `"air-condition"`. The "searchMode" parameter is used to match on any term (default) or all of them, for cases where a term isn't explicitly required (`+`).

3. Orders the resulting set of hotels by proximity to a given geography location, and then returns the results to the calling application.

Most this article is about processing of the *search query*: `"Spacious, air-condition* +\"Ocean view\""`. Filtering and ordering are out of scope. For more information, see the Search API reference documentation.

Stage 1: Query parsing

As noted, the query string is the first line of the request:

```
 "search": "Spacious, air-condition* +\"Ocean view\"",
```

The query parser separates operators (such as `*` and `+` in the example) from search terms, and deconstructs the search query into *subqueries* of a supported type:

- *term query* for standalone terms (like spacious)
- *phrase query* for quoted terms (like ocean view)
- *prefix query* for terms followed by a prefix operator `*` (like air-condition)

For a full list of supported query types see Lucene query syntax

Operators associated with a subquery determine whether the query "must be" or "should be" satisfied in order for a document to be considered a match. For example, `+"Ocean view"` is "must" due to the `+` operator.

The query parser restructures the subqueries into a *query tree* (an internal structure representing the query) it passes on to the search engine. In the first stage of query parsing, the query tree looks like this.

Supported parsers: Simple and Full Lucene

Azure Cognitive Search exposes two different query languages, `simple` (default) and `full`. By setting the `queryType` parameter with your search request, you tell the query parser which query language you choose so that it knows how to interpret the operators and syntax.

- The Simple query language is intuitive and robust, often suitable to interpret user input as-is without client-side processing. It supports query operators familiar from web search engines.
- The Full Lucene query language, which you get by setting `queryType=full`, extends the default Simple query language by adding support for more operators and query types like wildcard, fuzzy, regex, and field-scoped queries. For example, a regular expression sent in Simple query syntax would be interpreted as a query string and not an expression. The example request in this article uses the Full Lucene query language.

Impact of searchMode on the parser

Another search request parameter that affects parsing is the "searchMode" parameter. It controls the default operator for Boolean queries: any (default) or all.

When "searchMode=any", which is the default, the space delimiter between spacious and air-condition is OR (`||`), making the sample query text equivalent to:

```
Spacious,||air-condition*+"Ocean view"
```

Explicit operators, such as `+` in `+"Ocean view"`, are unambiguous in boolean query construction (the term *must* match). Less obvious is how to interpret the remaining terms: spacious and air-condition. Should the search engine find matches on ocean view *and* spacious *and* air-condition? Or should it find ocean view plus *either one* of the remaining terms?

By default ("searchMode=any"), the search engine assumes the broader interpretation. Either field *should* be matched, reflecting "or" semantics. The initial query tree illustrated previously, with the two "should" operations, shows the default.

Suppose that we now set "searchMode=all". In this case, the space is interpreted as an "and" operation. Each of the remaining terms must both be present in the document to qualify as a match. The resulting sample query would be interpreted as follows:

```
+Spacious,+air-condition*+"Ocean view"
```

A modified query tree for this query would be as follows, where a matching document is the intersection of all three subqueries:

> ⓘ **Note**
>
> Choosing "searchMode=any" over "searchMode=all" is a decision best arrived at by running representative queries. Users who are likely to include operators (common when searching document stores) might find results more intuitive if "searchMode=all" informs boolean query constructs. For more about the interplay between "searchMode" and operators, see **Simple query syntax**.

Stage 2: Lexical analysis

Lexical analyzers process *term queries* and *phrase queries* after the query tree is structured. An analyzer accepts the text inputs given to it by the parser, processes the

text, and then sends back tokenized terms to be incorporated into the query tree.

The most common form of lexical analysis is *linguistic analysis that transforms query terms based on rules specific to a given language:

- Reducing a query term to the root form of a word
- Removing non-essential words (stopwords, such as "the" or "and" in English)
- Breaking a composite word into component parts
- Lower casing an upper case word

All of these operations tend to erase differences between the text input provided by the user and the terms stored in the index. Such operations go beyond text processing and require in-depth knowledge of the language itself. To add this layer of linguistic awareness, Azure Cognitive Search supports a long list of language analyzers from both Lucene and Microsoft.

> ⓘ **Note**
>
> Analysis requirements can range from minimal to elaborate depending on your scenario. You can control complexity of lexical analysis by the selecting one of the predefined analyzers or by creating your own **custom analyzer**. Analyzers are scoped to searchable fields and are specified as part of a field definition. This allows you to vary lexical analysis on a per-field basis. Unspecified, the *standard* Lucene analyzer is used.

In our example, prior to analysis, the initial query tree has the term "Spacious," with an uppercase "S" and a comma that the query parser interprets as a part of the query term (a comma isn't considered a query language operator).

When the default analyzer processes the term, it will lowercase "ocean view" and "spacious", and remove the comma character. The modified query tree will look as follows:

Testing analyzer behaviors

The behavior of an analyzer can be tested using the Analyze API. Provide the text you want to analyze to see what terms given analyzer will generate. For example, to see how the standard analyzer would process the text "air-condition", you can issue the following request:

JSON

```
{
    "text": "air-condition",
    "analyzer": "standard"
}
```

The standard analyzer breaks the input text into the following two tokens, annotating them with attributes like start and end offsets (used for hit highlighting) as well as their position (used for phrase matching):

JSON

```
{
  "tokens": [
    {
      "token": "air",
      "startOffset": 0,
      "endOffset": 3,
      "position": 0
    },
    {
      "token": "condition",
      "startOffset": 4,
      "endOffset": 13,
      "position": 1
    }
  ]
}
```

Exceptions to lexical analysis

Lexical analysis applies only to query types that require complete terms – either a term query or a phrase query. It doesn't apply to query types with incomplete terms – prefix query, wildcard query, regex query – or to a fuzzy query. Those query types, including the prefix query with term `air-condition*` in our example, are added directly to the query tree, bypassing the analysis stage. The only transformation performed on query terms of those types is lowercasing.

Stage 3: Document retrieval

Document retrieval refers to finding documents with matching terms in the index. This stage is understood best through an example. Let's start with a hotels index having the following simple schema:

JSON

```
{
    "name": "hotels",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable":
false },
        { "name": "title", "type": "Edm.String", "searchable": true },
        { "name": "description", "type": "Edm.String", "searchable": true }
    ]
}
```

Further assume that this index contains the following four documents:

JSON

```
{
    "value": [
        {
            "id": "1",
            "title": "Hotel Atman",
            "description": "Spacious rooms, ocean view, walking distance to
the beach."
        },
        {
            "id": "2",
            "title": "Beach Resort",
            "description": "Located on the north shore of the island of
Kaua‘i. Ocean view."
        },
        {
            "id": "3",
            "title": "Playa Hotel",
            "description": "Comfortable, air-conditioned rooms with ocean
view."
        },
        {
            "id": "4",
            "title": "Ocean Retreat",
            "description": "Quiet and secluded"
        }
    ]
}
```

How terms are indexed

To understand retrieval, it helps to know a few basics about indexing. The unit of storage is an inverted index, one for each searchable field. Within an inverted index is a sorted list of all terms from all documents. Each term maps to the list of documents in which it occurs, as evident in the example below.

To produce the terms in an inverted index, the search engine performs lexical analysis over the content of documents, similar to what happens during query processing:

1. *Text inputs* are passed to an analyzer, lower-cased, stripped of punctuation, and so forth, depending on the analyzer configuration.
2. *Tokens* are the output of lexical analysis.
3. *Terms* are added to the index.

It's common, but not required, to use the same analyzers for search and indexing operations so that query terms look more like terms inside the index.

> ⓘ **Note**
>
> Azure Cognitive Search lets you specify different analyzers for indexing and search via additional `indexAnalyzer` and `searchAnalyzer` field parameters. If unspecified, the analyzer set with the `analyzer` property is used for both indexing and searching.

Inverted index for example documents

Returning to our example, for the **title** field, the inverted index looks like this:

Term	Document list
atman	1
beach	2
hotel	1, 3
ocean	4
playa	3
resort	3
retreat	4

In the title field, only *hotel* shows up in two documents: 1, 3.

For the **description** field, the index is as follows:

Term	Document list
air	3
and	4
beach	1
conditioned	3
comfortable	3
distance	1
island	2
kaua‘i	2
located	2
north	2
ocean	1, 2, 3
of	2
on	2
quiet	4
rooms	1, 3
secluded	4
shore	2
spacious	1
the	1, 2
to	1
view	1, 2, 3
walking	1
with	3

Matching query terms against indexed terms

Given the inverted indices above, let’s return to the sample query and see how matching documents are found for our example query. Recall that the final query tree looks like this:

During query execution, individual queries are executed against the searchable fields independently.

- The TermQuery, "spacious", matches document 1 (Hotel Atman).
- The PrefixQuery, "air-condition*", doesn't match any documents.

  This is a behavior that sometimes confuses developers. Although the term air-conditioned exists in the document, it's split into two terms by the default analyzer. Recall that prefix queries, which contain partial terms, aren't analyzed. Therefore terms with prefix "air-condition" are looked up in the inverted index and not found.
- The PhraseQuery, "ocean view", looks up the terms "ocean" and "view" and checks the proximity of terms in the original document. Documents 1, 2 and 3 match this query in the description field. Notice document 4 has the term ocean in the title but isn't considered a match, as we're looking for the "ocean view" phrase rather than individual words.

> **Note**
>
> A search query is executed independently against all searchable fields in the Azure Cognitive Search index unless you limit the fields set with the `searchFields` parameter, as illustrated in the example search request. Documents that match in any of the selected fields are returned.

On the whole, for the query in question, the documents that match are 1, 2, 3.

Stage 4: Scoring

Every document in a search result set is assigned a relevance score. The function of the relevance score is to rank higher those documents that best answer a user question as expressed by the search query. The score is computed based on statistical properties of terms that matched. At the core of the scoring formula is TF/IDF (term frequency-inverse document frequency). In queries containing rare and common terms, TF/IDF promotes results containing the rare term. For example, in a hypothetical index with all Wikipedia

articles, from documents that matched the query *the president*, documents matching on *president* are considered more relevant than documents matching on *the*.

Scoring example

Recall the three documents that matched our example query:

```
search=Spacious, air-condition* +"Ocean view"
```

JSON

```
{
  "value": [
    {
      "@search.score": 0.25610128,
      "id": "1",
      "title": "Hotel Atman",
      "description": "Spacious rooms, ocean view, walking distance to the
beach."
    },
    {
      "@search.score": 0.08951007,
      "id": "3",
      "title": "Playa Hotel",
      "description": "Comfortable, air-conditioned rooms with ocean view."
    },
    {
      "@search.score": 0.05967338,
      "id": "2",
      "title": "Ocean Resort",
      "description": "Located on a cliff on the north shore of the island of
Kauai. Ocean view."
    }
  ]
}
```

Document 1 matched the query best because both the term *spacious* and the required phrase *ocean view* occur in the description field. The next two documents match only the phrase *ocean view*. It might be surprising that the relevance score for document 2 and 3 is different even though they matched the query in the same way. It's because the scoring formula has more components than just TF/IDF. In this case, document 3 was assigned a slightly higher score because its description is shorter. Learn about Lucene's Practical Scoring Formula to understand how field length and other factors can influence the relevance score.

Some query types (wildcard, prefix, regex) always contribute a constant score to the overall document score. This allows matches found through query expansion to be included in the results, but without affecting the ranking.

An example illustrates why this matters. Wildcard searches, including prefix searches, are ambiguous by definition because the input is a partial string with potential matches on a very large number of disparate terms (consider an input of "tour*", with matches found on “tours”, “tourettes”, and “tourmaline”). Given the nature of these results, there's no way to reasonably infer which terms are more valuable than others. For this reason, we ignore term frequencies when scoring results in queries of types wildcard, prefix and regex. In a multi-part search request that includes partial and complete terms, results from the partial input are incorporated with a constant score to avoid bias towards potentially unexpected matches.

Relevance tuning

There are two ways to tune relevance scores in Azure Cognitive Search:

1. **Scoring profiles** promote documents in the ranked list of results based on a set of rules. In our example, we could consider documents that matched in the title field more relevant than documents that matched in the description field. Additionally, if our index had a price field for each hotel, we could promote documents with lower price. Learn more about adding scoring profiles to a search index.

2. **Term boosting** (available only in the Full Lucene query syntax) provides a boosting operator `^` that can be applied to any part of the query tree. In our example, instead of searching on the prefix *air-condition**, one could search for either the exact term *air-condition* or the prefix, but documents that match on the exact term are ranked higher by applying boost to the term query: *air-condition^2||air-condition**. Learn more about term boosting in a query.

Scoring in a distributed index

All indexes in Azure Cognitive Search are automatically split into multiple shards, allowing us to quickly distribute the index among multiple nodes during service scale up or scale down. When a search request is issued, it’s issued against each shard independently. The results from each shard are then merged and ordered by score (if no other ordering is defined). It's important to know that the scoring function weights query term frequency against its inverse document frequency in all documents within the shard, not across all shards!

This means a relevance score *could* be different for identical documents if they reside on different shards. Fortunately, such differences tend to disappear as the number of documents in the index grows due to more even term distribution. It’s not possible to assume on which shard any given document will be placed. However, assuming a document key doesn't change, it will always be assigned to the same shard.

In general, document score isn't the best attribute for ordering documents if order stability is important. For example, given two documents with an identical score, there's no guarantee which one appears first in subsequent runs of the same query. Document score should only give a general sense of document relevance relative to other documents in the results set.

Conclusion

The success of commercial search engines has raised expectations for full text search over private data. For almost any kind of search experience, we now expect the engine to understand our intent, even when terms are misspelled or incomplete. We might even expect matches based on near equivalent terms or synonyms that we never actually specified.

From a technical standpoint, full text search is highly complex, requiring sophisticated linguistic analysis and a systematic approach to processing in ways that distill, expand, and transform query terms to deliver a relevant result. Given the inherent complexities, there are many factors that can affect the outcome of a query. For this reason, investing the time to understand the mechanics of full text search offers tangible benefits when trying to work through unexpected results.

This article explored full text search in the context of Azure Cognitive Search. We hope it gives you sufficient background to recognize potential causes and resolutions for addressing common query problems.

Next steps

- Build the sample index, try out different queries and review results. For instructions, see Build and query an index in the portal.
- Try other query syntax from the Search Documents example section or from Simple query syntax in Search explorer in the portal.
- Review scoring profiles if you want to tune ranking in your search application.
- Learn how to apply language-specific lexical analyzers.

- Configure custom analyzers for either minimal processing or specialized processing on specific fields.

See also

Search Documents REST API

Simple query syntax

Full Lucene query syntax

Handle search results

Additional resources

Documentation

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Relevance and scoring - Azure Cognitive Search

Explains the concepts of relevance and scoring in Azure Cognitive Search, and what a developer can do to customize the scoring result.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to

search results in Azure Cognitive Search.

Show 5 more

Semantic search in Azure Cognitive Search

Article • 04/27/2023

> **Important**
>
> Semantic search is in public preview under **supplemental terms of use**. It's available through the Azure portal, preview REST API, and beta SDKs. These features are billable (see **Availability and pricing**).

Currently in Azure Cognitive Search, "semantic search" is a collection of query-related capabilities that bring semantic relevance and language understanding to textual search results. This article is a high-level introduction to semantic search. The embedded video describes the feature, and the section at the end covers availability and pricing.

Semantic search is a premium feature that's billed by usage. We recommend this article for background, but if you'd rather get started, follow these steps:

- ✓ Check regional and service tier requirements.
- ✓ Enable semantic search for semantic ranking on your search service.
- ✓ Create or modify queries to return semantic captions and highlights.
- ✓ Add a few more query properties to also return semantic answers.

What is semantic search?

Semantic search is a collection of features that improve the quality of initial search results for text-based queries. When you enable it on your search service, semantic search extends the query execution pipeline in two ways:

- First, it adds secondary ranking over an initial result set, promoting the most semantically relevant results to the top of the list.
- Second, it extracts and returns captions and answers in the response, which you can render on a search page to improve the user's search experience.

Although semantic search and vector search are closely related, this particular feature doesn't provide vector search.

Feature	Description

Feature	Description
Semantic re-ranking	Uses the context or semantic meaning of a query to compute a new relevance score over existing results.
Semantic captions and highlights	Extracts sentences and phrases from a document that best summarize the content, with highlights over key passages for easy scanning. Captions that summarize a result are useful when individual content fields are too dense for the results page. Highlighted text elevates the most relevant terms and phrases so that users can quickly determine why a match was considered relevant.
Semantic answers	An optional and extra substructure returned from a semantic query. It provides a direct answer to a query that looks like a question. It requires that a document has text with the characteristics of an answer.

How semantic ranking works

Semantic ranking looks for context and relatedness among terms, elevating matches that make more sense given the query. Language understanding finds summarizations or *captions* and *answers* within your content and includes them in the response, which can then be rendered on a search results page for a more productive search experience.

State-of-the-art pretrained models are used for summarization and ranking. To maintain the fast performance that users expect from search, semantic summarization and ranking are applied to just the top 50 results, as scored by the default scoring algorithm. Using those results as the document corpus, semantic ranking re-scores those results based on the semantic strength of the match.

The underlying technology is from Bing and Microsoft Research, and integrated into the Cognitive Search infrastructure as an add-on feature. For more information about the research and AI investments backing semantic search, see How AI from Bing is powering Azure Cognitive Search (Microsoft Research Blog).

The following video provides an overview of the capabilities.
https://www.youtube-nocookie.com/embed/yOf0WfVd_V0

Order of operations

Components of semantic search extend the existing query execution pipeline in both directions. If you enable spelling correction, the speller corrects typos at query onset, before terms reach the search engine.

Query execution proceeds as usual, with term parsing, analysis, and scans over the inverted indexes. The engine retrieves documents using token matching, and scores the results using the default scoring algorithm. Scores are calculated based on the degree of linguistic similarity between query terms and matching terms in the index. If you defined them, scoring profiles are also applied at this stage. Results are then passed to the semantic search subsystem.

In the preparation step, the document corpus returned from the initial result set is analyzed at the sentence and paragraph level to find passages that summarize each document. In contrast with keyword search, this step uses machine reading and comprehension to evaluate the content. Through this stage of content processing, a semantic query returns captions and answers. To formulate them, semantic search uses language representation to extract and highlight key passages that best summarize a result. If the search query is a question - and answers are requested - the response will also include a text passage that best answers the question, as expressed by the search query.

For both captions and answers, existing text is used in the formulation. The semantic models don't compose new sentences or phrases from the available content, nor does it apply logic to arrive at new conclusions. In short, the system will never return content that doesn't already exist.

Results are then re-scored based on the conceptual similarity of query terms.

To use semantic capabilities in queries, you'll need to make small modifications to the search request, but no extra configuration or reindexing is required.

Semantic capabilities and limitations

Semantic search is a newer technology so it's important to set expectations about what it can and can't do. What it can do is improve the quality of search by:

- Promoting matches that are semantically closer to the intent of original query.
- Finding strings in each result that can be used as captions, and potentially answers, that can be rendered on a search results page.

What it can't do is rerun the query over the entire corpus to find semantically relevant results. Semantic search re-ranks the *existing* result set, consisting of the top 50 results as scored by the default ranking algorithm. Furthermore, semantic search can't create new information or strings. Captions and answers are extracted verbatim from your content so if the results don't include answer-like text, the language models won't produce one.

Although semantic search isn't beneficial in every scenario, certain content can benefit significantly from its capabilities. The language models in semantic search work best on searchable content that is information-rich and structured as prose. A knowledge base, online documentation, or documents that contain descriptive content see the most gains from semantic search capabilities.

Availability and pricing

Semantic search and spell check are available on services that meet the criteria in the table below. To use semantic search, your first need to enable the capabilities on your search service.

Feature	Tier	Region	Sign up	Pricing
Semantic search	Standard tier (S1, S2, S3, S3 HD), Storage Optimized tier (L1, L2)	Region availability	Required	Pricing [1]
Spell check	Basic [2] and above	All	None	None (free)

[1] On the pricing page, scroll down to view more features that are billed separately. At lower query volumes (under 1000 monthly), semantic search is free. To go above that limit, you can opt in to the semantic search standard pricing plan. The pricing page shows you the semantic query billing rate for different currencies and intervals.

[2] Due to the provisioning mechanisms and lifespan of shared (free) search services, a few services happen to have spell check on the free tier. However, spell check availability on free tier services isn't guaranteed and shouldn't be expected.

Charges for semantic search are levied when query requests include "queryType=semantic" and the search string isn't empty (for example, "search=pet friendly hotels in New York". If your search string is empty ("search=*"), you won't be charged, even if the queryType is set to "semantic".

Enable semantic search

By default, semantic search is disabled on all services. To enable semantic search for your search service:

1. Open the Azure portal.
2. Navigate to your Standard tier search service.
3. Determine whether the service region supports semantic search. Search service region is noted on the overview page. Semantic search regions are noted on the Products Available by Region page.
4. On the left-nav pane, select **Semantic Search (Preview)**.
5. Select either the **Free plan** or the **Standard plan**. You can switch between the free plan and the standard plan at any time.

Semantic Search's free plan is capped at 1,000 queries per month. After the first 1,000 queries in the free plan, you'll receive an error message letting you know you've exhausted your quota whenever you issue a semantic query. When this happens, you need to upgrade to the standard plan to continue using semantic search.

Alternatively, you can also enable semantic search using the Create or Update Service API that's described in the next section.

Disable semantic search

To reverse feature enablement, or for full protection against accidental usage and charges, you can disable semantic search using the Create or Update Service API on

your search service. After the feature is disabled, any requests that include the semantic query type will be rejected.

- Management REST API version 2021-04-01-Preview provides this option
- Owner or Contributor permissions are required to disable features

HTTP

```
PUT
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2021-04-01-Preview
    {
      "location": "{{region}}",
      "sku": {
        "name": "standard"
      },
      "properties": {
        "semanticSearch": "disabled"
      }
    }
```

To re-enable semantic search, rerun the above request, setting "semanticSearch" to either "free" (default) or "standard".

> **Tip**
>
> Management REST API calls are authenticated through Azure Active Directory. For guidance on setting up a security principal and a request, see this blog post **Azure REST APIs with Postman (2021)**. The previous example was tested using the instructions and Postman collection provided in the blog post.

Next steps

Enable semantic search for your search service and follow the steps in Configure semantic ranking so that you can test out semantic search on your content.

Querying in Azure Cognitive Search

Article • 03/01/2023 • 7 minutes to read

Azure Cognitive Search offers a rich query language to support a broad range of scenarios, from free text search, to highly-specified query patterns. This article describes query requests and the kinds of queries you can create.

In Cognitive Search, a query is a full specification of a round-trip `search` operation, with parameters that both inform query execution and shape the response coming back. To illustrate, the following query example calls the Search Documents (REST API). It's a parameterized, free text query with a boolean operator, targeting the hotels-sample-index documents collection. It also selects which fields are returned in results.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "queryType": "simple",
    "searchMode": "all",
    "search": "restaurant +view",
    "searchFields": "HotelName, Description, Address/City, Address/StateProvince, Tags",
    "select": "HotelName, Description, Address/City, Address/StateProvince, Tags",
    "top": "10",
    "count": "true",
    "orderby": "Rating desc"
}
```

Parameters used during query execution include:

- `queryType` sets the parser: `simple`, `full`. The default simple query parser is optimal for full text search. The full Lucene query parser is for advanced query constructs like regular expressions, proximity search, fuzzy and wildcard search. This parameter can also be set to `semantic` for semantic search for advanced semantic modeling on the query response.

- `searchMode` specifies whether matches are based on "all" criteria (favors precision) or "any" criteria (favors recall) in the expression. The default is "any".

- `search` provides the match criteria, usually whole terms or phrases, with or without operators. Any field that is attributed as "searchable" in the index schema is a candidate for this parameter.

- `searchFields` constrains query execution to specific searchable fields. During development, it's helpful to use the same field list for select and search. Otherwise a match might be based on field values that you can't see in the results, creating uncertainty as to why the document was returned.

Parameters used to shape the response:

- `select` specifies which fields to return in the response. Only fields marked as "retrievable" in the index can be used in a select statement.
- `top` returns the specified number of best-matching documents. In this example, only 10 hits are returned. You can use top and skip (not shown) to page the results.
- `count` tells you how many documents in the entire index match overall, which can be more than what are returned.
- `orderby` is used if you want to sort results by a value, such as a rating or location. Otherwise, the default is to use the relevance score to rank results. A field must be attributed as "sortable" to be a candidate for this parameter.

The above list is representative but not exhaustive. For the full list of parameters on a query request, see Search Documents (REST API).

Types of queries

With a few notable exceptions, a query request iterates over inverted indexes that are structured for fast scans, where a match can be found in potentially any field, within any number of search documents. In Cognitive Search, the primary methodology for finding matches is either full text search or filters, but you can also implement other well-known search experiences like autocomplete, or geo-location search. The rest of this article summarizes queries in Cognitive Search and provides links to more information and examples.

Full text search

If your search app includes a search box that collects term inputs, then full text search is probably the query operation backing that experience. Full text search accepts terms or phrases passed in a `search` parameter in all "searchable" fields in your index. Optional boolean operators in the query string can specify inclusion or exclusion criteria. Both the simple parser and full parser support full text search.

In Cognitive Search, full text search is built on the Apache Lucene query engine. Query strings in full text search undergo lexical analysis to make scans more efficient. Analysis includes lower-casing all terms, removing stop words like "the", and reducing terms to primitive root forms. The default analyzer is Standard Lucene.

When matching terms are found, the query engine reconstitutes a search document containing the match using the document key or ID to assemble field values, ranks the documents in order of relevance, and returns the top 50 (by default) in the response or a different number if you specified `top`.

If you're implementing full text search, understanding how your content is tokenized will help you debug any query anomalies. Queries over hyphenated strings or special characters could necessitate using an analyzer other than the default standard Lucene to ensure the index contains the right tokens. You can override the default with language analyzers or specialized analyzers that modify lexical analysis. One example is keyword that treats the entire contents of a field as a single token. This is useful for data like zip codes, IDs, and some product names. For more information, see Partial term search and patterns with special characters.

> **Tip**
>
> If you anticipate heavy use of Boolean operators, which is more likely in indexes that contain large text blocks (a content field or long descriptions), be sure to test queries with the `searchMode=Any|All` parameter to evaluate the impact of that setting on boolean search.

Autocomplete and suggested queries

Autocomplete or suggested results are alternatives to `search` that fire successive query requests based on partial string inputs (after each character) in a search-as-you-type experience. You can use `autocomplete` and `suggestions` parameter together or separately, as described in this tutorial, but you cannot use them with `search`. Both completed terms and suggested queries are derived from index contents. The engine will never return a string or suggestion that is non-existent in your index. For more information, see Autocomplete (REST API) and Suggestions (REST API).

Filter search

Filters are widely used in apps that are based on Cognitive Search. On application pages, filters are often visualized as facets in link navigation structures for user-directed

filtering. Filters are also used internally to expose slices of indexed content. For example, you might initialize a search page using a filter on a product category, or a language if an index contains fields in both English and French.

You might also need filters to invoke a specialized query form, as described in the following table. You can use a filter with an unspecified search (`search=*`) or with a query string that includes terms, phrases, operators, and patterns.

Filter scenario	Description
Range filters	In Azure Cognitive Search, range queries are built using the filter parameter. For more information and examples, see Range filter example.
Faceted navigation	In faceted navigation tree, users can select facets. When backed by filters, search results narrow on each click. Each facet is backed by a filter that excludes documents that no longer match the criteria provided by the facet.

> ⓘ **Note**
>
> Text that's used in a filter expression is not analyzed during query processing. The text input is presumed to be a verbatim case-sensitive character pattern that either succeeds or fails on the match. Filter expressions are constructed using **OData syntax** and passed in a `filter` parameter in all *filterable* fields in your index. For more information, see **Filters in Azure Cognitive Search**.

Geospatial search

Geospatial search matches on a location's latitude and longitude coordinates for "find near me" or map-based search experience. In Azure Cognitive Search, you can implement geospatial search by following these steps:

- Define a filterable field of one of these types: Edm.GeographyPoint, Collection(Edm.GeographyPoint, Edm.GeographyPolygon).
- Verify the incoming documents include the appropriate coordinates.
- After indexing is complete, build a query that uses a filter and a geo-spatial function.

For more information and an example, see Geospatial search example.

Document look-up

In contrast with the previously described query forms, this one retrieves a single search document by ID, with no corresponding index search or scan. Only the one document is requested and returned. When a user selects an item in search results, retrieving the document and populating a details page with fields is a typical response, and a document look-up is the operation that supports it.

Advanced search: fuzzy, wildcard, proximity, regex

An advanced query form depends on the Full Lucene parser and operators that trigger a specific query behavior.

Query type	Usage	Examples and more information
Fielded search	`search` parameter, `queryType=full`	Build a composite query expression targeting a single field. Fielded search example
fuzzy search	`search` parameter, `queryType=full`	Matches on terms having a similar construction or spelling. Fuzzy search example
proximity search	`search` parameter, `queryType=full`	Finds terms that are near each other in a document. Proximity search example
term boosting	`search` parameter, `queryType=full`	Ranks a document higher if it contains the boosted term, relative to others that don't. Term boosting example
regular expression search	`search` parameter, `queryType=full`	Matches based on the contents of a regular expression. Regular expression example
wildcard or prefix search	`search` parameter with *`~` or `?`, `queryType=full`	Matches based on a prefix and tilde (`~`) or single character (`?`). Wildcard search example

Next steps

For a closer look at query implementation, review the examples for each syntax. If you are new to full text search, a closer look at what the query engine does might be an equally good choice.

- Simple query examples

- Lucene syntax query examples for building advanced queries
- How full text search works in Azure Cognitive Searchgit

Additional resources

🕮 Documentation

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Add scoring profiles to boost search scores - Azure Cognitive Search

Boost search relevance scores for Azure Cognitive Search results by adding scoring profiles to a search index.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Add a faceted navigation category hierarchy - Azure Cognitive Search

Add faceted navigation for self-directed filtering in applications that integrate with Azure Cognitive Search.

Create Index (Azure Cognitive Search REST API)

Define an index schema for fields and other constructs in an Azure Cognitive Search index.

Show 4 more

Relevance and scoring in Azure Cognitive Search

Article • 04/18/2023

This article explains the relevance and the scoring algorithms used to compute search scores in Azure Cognitive Search. A relevance score is computed for each match found in a full text search, where the strongest matches are assigned higher search scores.

Relevance applies to full text search only. Filter queries, autocomplete and suggested queries, wildcard search or fuzzy search queries are not scored or ranked for relevance.

In Azure Cognitive Search, you can tune search relevance and boost search scores through these mechanisms:

- Scoring algorithm configuration
- Semantic ranking (in preview, described in this article)
- Scoring profiles
- Custom scoring logic enabled through the *featuresMode* parameter

> **Note**
>
> Matches are scored and ranked from high to low. The score is returned as "@search.score". By default, the top 50 are returned in the response, but you can use the **$top** parameter to return a smaller or larger number of items (up to 1000 in a single response), and **$skip** to get the next set of results.

Relevance scoring

Relevance scoring refers to the computation of a search score that serves as an indicator of an item's relevance in the context of the current query. The higher the score, the more relevant the item.

The search score is computed based on statistical properties of the string input and the query itself. Azure Cognitive Search finds documents that match on search terms (some or all, depending on searchMode), favoring documents that contain many instances of the search term. The search score goes up even higher if the term is rare across the data index, but common within the document. The basis for this approach to computing relevance is known as *TF-IDF or* term frequency-inverse document frequency.

Search scores can be repeated throughout a result set. When multiple hits have the same search score, the ordering of the same scored items is undefined and not stable. Run the query again, and you might see items shift position, especially if you are using the free service or a billable service with multiple replicas. Given two items with an identical score, there is no guarantee which one appears first.

If you want to break the tie among repeating scores, you can add an **$orderby** clause to first order by score, then order by another sortable field (for example, `$orderby=search.score() desc,Rating desc`). For more information, see $orderby.

> ⓘ **Note**
>
> A `@search.score = 1` indicates an un-scored or un-ranked result set. The score is uniform across all results. Un-scored results occur when the query form is fuzzy search, wildcard or regex queries, or an empty search (`search=*`, sometimes paired with filters, where the filter is the primary means for returning a match).

Scoring algorithms in Search

Azure Cognitive Search provides the following scoring algorithms:

Algorithm	Usage	Range
BM25Similarity	Fixed algorithm on all search services created after July 2020. You can configure this algorithm, but you can't switch to an older one (classic).	Unbounded.
ClassicSimilarity	Present on older search services. You can opt-in for BM25 and choose an algorithm on a per-index basis.	0 < 1.00

Both BM25 and Classic are TF-IDF-like retrieval functions that use the term frequency (TF) and the inverse document frequency (IDF) as variables to calculate relevance scores for each document-query pair, which is then used for ranking results. While conceptually similar to classic, BM25 is rooted in probabilistic information retrieval that produces more intuitive matches, as measured by user research.

BM25 offers advanced customization options, such as allowing the user to decide how the relevance score scales with the term frequency of matched terms. For more information, see Configure the scoring algorithm.

> ⓘ **Note**

If you're using a search service that was created before July 2020, the scoring algorithm is most likely the previous default, `ClassicSimilarity`, which you can upgrade on a per-index basis. See **Enable BM25 scoring on older services** for details.

The following video segment fast-forwards to an explanation of the generally available ranking algorithms used in Azure Cognitive Search. You can watch the full video for more background.
https://www.youtube-nocookie.com/embed/Y_X6USgvB1g?version=3&start=322&end=643

Score variation

Search scores convey general sense of relevance, reflecting the strength of match relative to other documents in the same result set. But scores aren't always consistent from one query to the next, so as you work with queries, you might notice small discrepancies in how search documents are ordered. There are several explanations for why this might occur.

Cause	Description
Data volatility	Index content varies as you add, modify, or delete documents. Term frequencies will change as index updates are processed over time, affecting the search scores of matching documents.
Multiple replicas	For services using multiple replicas, queries are issued against each replica in parallel. The index statistics used to calculate a search score are calculated on a per-replica basis, with results merged and ordered in the query response. Replicas are mostly mirrors of each other, but statistics can differ due to small differences in state. For example, one replica might have deleted documents contributing to their statistics, which were merged out of other replicas. Typically, differences in per-replica statistics are more noticeable in smaller indexes. For more information about this condition, see Concepts: search units, replicas, partitions, shards in the capacity planning documentation.
Identical scores	If multiple documents have the same score, any one of them might appear first.

Scoring statistics and sticky sessions

For scalability, Azure Cognitive Search distributes each index horizontally through a sharding process, which means that portions of an index are physically separate.

By default, the score of a document is calculated based on statistical properties of the data *within a shard*. This approach is generally not a problem for a large corpus of data, and it provides better performance than having to calculate the score based on information across all shards. That said, using this performance optimization could cause two very similar documents (or even identical documents) to end up with different relevance scores if they end up in different shards.

If you prefer to compute the score based on the statistical properties across all shards, you can do so by adding *scoringStatistics=global* as a query parameter (or add *"scoringStatistics": "global"* as a body parameter of the query request).

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?
api-version=2020-06-30
{
    "search": "<query string>",
    "scoringStatistics": "global"
}
```

Using scoringStatistics will ensure that all shards in the same replica provide the same results. That said, different replicas may be slightly different from one another as they are always getting updated with the latest changes to your index. In some scenarios, you may want your users to get more consistent results during a "query session". In such scenarios, you can provide a `sessionId` as part of your queries. The `sessionId` is a unique string that you create to refer to a unique user session.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?
api-version=2020-06-30
{
    "search": "<query string>",
    "sessionId": "<string>"
}
```

As long as the same `sessionId` is used, a best-effort attempt will be made to target the same replica, increasing the consistency of results your users will see.

Note

Reusing the same `sessionId` values repeatedly can interfere with the load balancing of the requests across replicas and adversely affect the performance of the search service. The value used as sessionId cannot start with a '_' character.

Scoring profiles

You can customize the way different fields are ranked by defining a *scoring profile*. Scoring profiles provide criteria for boosting the search score of a match based on content characteristics. For example, you might want to boost matches based on their revenue potential, promote newer items, or perhaps boost items that have been in inventory too long.

A scoring profile is part of the index definition, composed of weighted fields, functions, and parameters. For more information about defining one, see Scoring Profiles.

featuresMode parameter (preview)

Search Documents requests have a new featuresMode parameter that can provide additional detail about relevance at the field level. Whereas the `@searchScore` is calculated for the document all-up (how relevant is this document in the context of this query), through featuresMode you can get information about individual fields, as expressed in a `@search.features` structure. The structure contains all fields used in the query (either specific fields through **searchFields** in a query, or all fields attributed as **searchable** in an index). For each field, you get the following values:

- Number of unique tokens found in the field
- Similarity score, or a measure of how similar the content of the field is, relative to the query term
- Term frequency, or the number of times the query term was found in the field

For a query that targets the "description" and "title" fields, a response that includes `@search.features` might look like this:

JSON

```
"value": [
 {
    "@search.score": 5.1958685,
    "@search.features": {
        "description": {
            "uniqueTokenMatches": 1.0,
            "similarityScore": 0.29541412,
            "termFrequency" : 2
        },
        "title": {
            "uniqueTokenMatches": 3.0,
            "similarityScore": 1.75451557,
```

```
            "termFrequency" : 6
        }
```

You can consume these data points in custom scoring solutions or use the information to debug search relevance problems.

See also

- Scoring Profiles
- REST API Reference
- Search Documents API
- Azure Cognitive Search .NET SDK

Semantic ranking in Azure Cognitive Search

Article • 11/02/2022 • 4 minutes to read

> **Important**
>
> Semantic search is in public preview under **supplemental terms of use**. It's available through the Azure portal, preview REST API, and beta SDKs. These features are billable. For more information about, see **Availability and pricing**.

Semantic ranking is an extension of the query execution pipeline that improves precision by reranking the top matches of an initial result set. Semantic ranking is backed by large transformer-based networks, trained for capturing the semantic meaning of query terms, as opposed to linguistic matching on keywords. In contrast with the default similarity ranking algorithm, the semantic ranker uses the context and meaning of words to determine relevance.

Semantic ranking is both resource and time intensive. In order to complete processing within the expected latency of a query operation, inputs to the semantic ranker are consolidated and reduced so that the underlying summarization and reranking steps can be completed as quickly as possible.

Pre-processing

Before scoring for relevance, content must be reduced to a manageable number of inputs that can be handled efficiently by the semantic ranker.

1. First, content reduction begins with the initial results set returned by the default similarity ranking algorithm used for keyword search. For any given query, the results could be a handful of documents, up to the maximum limit of 1,000. Because processing a large number of matches would take too long, only the top 50 progress to semantic ranking.

   Whatever the document count, whether one or 50, the initial result set establishes the first iteration of the document corpus for semantic ranking.

2. Next, across the corpus, the contents of each field in the semantic configuration are extracted and combined into a long string.

3. After string consolidation, any strings that are excessively long are trimmed to ensure the overall length meets the input requirements of the summarization step.

   This trimming exercise is why it's important to add fields to your semantic configuration in prioritized order. If you have very large documents with text-heavy fields, anything after the maximum limit is ignored.

Each document is now represented by a single long string.

> **Note**
>
> In the 2020-06-30-preview, the "searchFields" parameter is used rather than the semantic configuration to determine which fields to use. We recommend upgrading to the 2021-04-30-preview API version for best results.

The string is composed of tokens, not characters or words. The maximum token count is 128 unique tokens. For estimation purposes, you can assume that 128 tokens are roughly equivalent to a string that is 128 words in length.

> **Note**
>
> Tokenization is determined in part by the analyzer assignment on searchable fields. If you are using specialized analyzer, such as nGram or EdgeNGram, you might want to exclude that field from searchFields. For insights into how strings are tokenized, you can review the token output of an analyzer using the **Test Analyzer REST API**.

Extraction

After string reduction, it's now possible to pass the reduced inputs through machine reading comprehension and language representation models to determine which sentences and phrases best summarize the document, relative to the query. This phase extracts content from the string that will move forward to the semantic ranker.

Inputs to summarization are the long strings obtained for each document in the preparation phase. From each string, the summarization model finds a passage that is the most representative. This passage also constitutes a semantic caption for the document. Each caption is available in a plain text version and a highlight version, and is frequently fewer than 200 words per document.

A semantic answer will also be returned if you specified the "answers" parameter, if the query was posed as a question, and if a passage can be found in the long string that is likely to provide an answer to the question.

Semantic ranking

1. Captions are evaluated for conceptual and semantic relevance, relative to the query provided.

   The following diagram provides an illustration of what "semantic relevance" means. Consider the term "capital", which could be used in the context of finance, law, geography, or grammar. If a query includes terms from the same vector space (for example, "capital" and "investment"), a document that also includes tokens in the same cluster will score higher than one that doesn't.

2. The @search.rerankerScore is assigned to each document based on the semantic relevance of the caption. Scores range from 4 to 0 (high to low), where a higher score indicates a stronger match.

3. After all documents are scored, they're listed in descending order by score and included in the query response payload. The payload includes answers, plain text and highlighted captions, and any fields that you marked as retrievable or specified in a select clause.

Next steps

Semantic ranking is offered on Standard tiers, in specific regions. For more information about availability and sign-up, see Availability and pricing. A new query type enables the ranking and response structures of semantic search. To get started, Configure semantic ranking.

Alternatively, review the following articles about default ranking. Semantic ranking depends on the similarity ranker to return the initial results. Knowing about query execution and ranking will give you a broad understanding of how the entire process works.

- Full text search in Azure Cognitive Search
- Similarity and scoring in Azure Cognitive Search
- Analyzers for text processing in Azure Cognitive Search

Indexes in Azure Cognitive Search

Article • 05/11/2022 • 10 minutes to read

In Azure Cognitive Search, a *search index* is your searchable content, available to the search engine for indexing, full text search, and filtered queries. An index is defined by a schema and saved to the search service, with data import following as a second step. This content exists within your search service, apart from your primary data stores, which is necessary for the millisecond response times expected in modern applications. Except for specific indexing scenarios, the search service will never connect to or query your local data.

If you're creating and managing a search index, this article will help you understand the following:

- Content (documents and schema)
- Physical representation
- Basic operations

Prefer to be hands-on right away? See Create a search index instead.

Content of a search index

In Cognitive Search, indexes contain *search documents*. Conceptually, a document is a single unit of searchable data in your index. For example, a retailer might have a document for each product, a news organization might have a document for each article, a travel site might have a document for each hotel and destination, and so forth. Mapping these concepts to more familiar database equivalents: a *search index* equates to a *table*, and *documents* are roughly equivalent to *rows* in a table.

The structure of a document is determined by the index schema, as illustrated below. The "fields" collection is typically the largest part of an index, where each field is named, assigned a data type, and attributed with allowable behaviors that determine how it is used.

JSON

```
{
  "name": "name_of_index, unique across the service",
  "fields": [
    {
      "name": "name_of_field",
      "type": "Edm.String | Collection(Edm.String) | Edm.Int32 | Edm.Int64 |
Edm.Double | Edm.Boolean | Edm.DateTimeOffset | Edm.GeographyPoint",
```

```
      "searchable": true (default where applicable) | false (only Edm.String
and Collection(Edm.String) fields can be searchable),
      "filterable": true (default) | false,
      "sortable": true (default where applicable) | false
(Collection(Edm.String) fields cannot be sortable),
      "facetable": true (default where applicable) | false
(Edm.GeographyPoint fields cannot be facetable),
      "key": true | false (default, only Edm.String fields can be keys),
      "retrievable": true (default) | false,
      "analyzer": "name_of_analyzer_for_search_and_indexing", (only if
'searchAnalyzer' and 'indexAnalyzer' are not set)
      "searchAnalyzer": "name_of_search_analyzer", (only if 'indexAnalyzer'
is set and 'analyzer' is not set)
      "indexAnalyzer": "name_of_indexing_analyzer", (only if
'searchAnalyzer' is set and 'analyzer' is not set)
      "synonymMaps": [ "name_of_synonym_map" ] (optional, only one synonym
map per field is currently supported)
    }
  ],
  "suggesters": [ ],
  "scoringProfiles": [ ],
  "analyzers":(optional)[ ... ],
  "charFilters":(optional)[ ... ],
  "tokenizers":(optional)[ ... ],
  "tokenFilters":(optional)[ ... ],
  "defaultScoringProfile": (optional) "...",
  "corsOptions": (optional) { },
  "encryptionKey":(optional){ }
  }
}
```

Other elements are collapsed for brevity, but the following links can provide the detail:

- Suggesters support type-ahead queries like autocomplete
- Scoring Profiles are used for relevance tuning
- Analyzers are used to process strings into tokens according to linguistic rules or other characteristics supported by the analyzer
- Cross-origin remote scripting (CORS) is used for apps that issues requests from different domains
- Encryption key is used for double-encryption of sensitive content in the index.

Field definitions

A search document is defined by the "fields" collection in the body of Create Index request. You will need fields for document identification (keys), storing searchable text, and fields for supporting filters, facets, and sorts. You might also need fields for data that a user never sees. For example, you might want fields for profit margins or marketing promotions that you can use to modify search rank.

If incoming data is hierarchical in nature, you can represent it within an index as a complex type, used to represent nested structures. The built-in sample data set, Hotels, illustrates complex types using an Address (contains multiple sub-fields) that has a one-to-one relationship with each hotel, and a Rooms complex collection, where multiple rooms are associated with each hotel.

Field attributes

Field attributes determine how a field is used, such as whether it is used in full text search, faceted navigation, sort operations, and so forth.

String fields are often marked as "searchable" and "retrievable". Fields used to narrow search results include "sortable", "filterable", and "facetable".

Attribute	Description
"searchable"	Full-text searchable, subject to lexical analysis such as word-breaking during indexing. If you set a searchable field to a value like "sunny day", internally it will be split into the individual tokens "sunny" and "day". For details, see How full text search works.
"filterable"	Referenced in $filter queries. Filterable fields of type `Edm.String` or `Collection(Edm.String)` do not undergo word-breaking, so comparisons are for exact matches only. For example, if you set such a field f to "sunny day", `$filter=f eq 'sunny'` will find no matches, but `$filter=f eq 'sunny day'` will.
"sortable"	By default the system sorts results by score, but you can configure sort based on fields in the documents. Fields of type `Collection(Edm.String)` cannot be "sortable".
"facetable"	Typically used in a presentation of search results that includes a hit count by category (for example, hotels in a specific city). This option cannot be used with fields of type `Edm.GeographyPoint`. Fields of type `Edm.String` that are filterable, "sortable", or "facetable" can be at most 32 kilobytes in length. For details, see Create Index (REST API).
"key"	Unique identifier for documents within the index. Exactly one field must be chosen as the key field and it must be of type `Edm.String`.
"retrievable"	Determines whether the field can be returned in a search result. This is useful when you want to use a field (such as *profit margin*) as a filter, sorting, or scoring mechanism, but do not want the field to be visible to the end user. This attribute must be `true` for `key` fields.

Although you can add new fields at any time, existing field definitions are locked in for the lifetime of the index. For this reason, developers typically use the portal for creating simple indexes, testing ideas, or using the portal pages to look up a setting. Frequent

iteration over an index design is more efficient if you follow a code-based approach so that you can rebuild the index easily.

> ⓘ **Note**
>
> The APIs you use to build an index have varying default behaviors. For the **REST APIs**, most attributes are enabled by default (for example, "searchable" and "retrievable" are true for string fields) and you often only need to set them if you want to turn them off. For the .NET SDK, the opposite is true. On any property you do not explicitly set, the default is to disable the corresponding search behavior unless you specifically enable it.

Physical structure and size

In Azure Cognitive Search, the physical structure of an index is largely an internal implementation. You can access its schema, query its content, monitor its size, and manage capacity, but the clusters themselves (indices, shards, and other files and folders) are managed internally by Microsoft.

You can monitor index size in the Indexes tab in the Azure portal, or by issuing a GET INDEX request against your search service. You can also issue a Service Statistics request and check the value of storage size.

The size of an index is determined by:

- Quantity and composition of your documents
- Attributes on individual fields
- Index configuration (specifically, whether you include suggesters)

Document composition and quantity is determined by what you choose to import. Remember that a search index should only contain searchable content. If source data includes binary fields, omit those fields unless you are using AI enrichment to crack and analyze the content to create text searchable information.

Field attributes determine behaviors. To support those behaviors, the indexing process creates the necessary data structures. For example, "searchable" invokes full text search, which scans inverted indices for the tokenized term. In contrast, a "filterable" or "sortable" attribute supports iteration over unmodified strings. The example in the next section shows variations in index size based on the selected attributes.

Suggesters are constructs that support type-ahead or autocomplete queries. As such, when you include a suggester, the indexing process will create the data structures

necessary for verbatim character matches. Suggesters are implemented at the field level, so choose only those fields that are reasonable for type-ahead.

Example demonstrating the storage implications of attributes and suggesters

The following screenshot illustrates index storage patterns resulting from various combinations of attributes. The index is based on the **real estate sample index**, which you can create easily using the Import data wizard and built-in sample data. Although the index schemas are not shown, you can infer the attributes based on the index name. For example, *realestate-searchable* index has the "searchable" attribute selected and nothing else, *realestate-retrievable* index has the "retrievable" attribute selected and nothing else, and so forth.

NAME	DOCUMENT COUNT	STORAGE SIZE
realestate-all-attributes-no-suggester	4,959	26.55 MiB
realestate-all-attributes-plus-suggester	4,959	49.4 MiB
realestate-filterable-facetable-sortable	4,959	20.89 MiB
realestate-no-attributes	4,959	4.99 MiB
realestate-retrievable	4,959	5.04 MiB
realestate-searchable	4,959	9.95 MiB

Although these index variants are somewhat artificial, we can refer to them for broad comparisons of how attributes affect storage:

- "retrievable" has no impact on index size.
- "filterable", "sortable", "facetable" consume more storage.
- **suggester** has a large potential for increasing index size, but not as much as the screenshot would indicate (all fields that could be made suggester-aware were selected, which isn't a likely scenario in most indexes).

Also not reflected in the above table is the impact of analyzers. If you are using the edgeNgram tokenizer to store verbatim sequences of characters (a, ab, abc, abcd), the size of the index will be larger than if you used a standard analyzer.

Basic operations and interaction

Now that you have a better idea of what an index is, this section introduces index run time operations, including connecting to and securing a single index.

> **Note**
>
> When managing an index, be aware that there is no portal or API support for moving or copying an index. Instead, customers typically point their application deployment solution at a different search service (if using the same index name), or revise the name to create a copy on the current search service, and then build it.

Index isolation

In Cognitive Search, you'll work with one index at a time, where all index-related operations target a single index. There is no concept of related indexes or the joining of independent indexes for either indexing or querying.

Continuously available

An index is continuously available, with no ability to pause or take it offline. Because it's designed for continuous operation, any updates to its content, or additions to the index itself, happen in real time. As a result, queries might temporarily return incomplete results if a request coincides with a document update.

Notice that query continuity exists for document operations (refreshing or deleting) and for modifications that don't impact the existing structure and integrity of the current index (such as adding new fields). If you need to make structural updates (changing existing fields), those are typically managed using a drop-and-rebuild workflow in a development environment, or by creating a new version of the index on production service.

To avoid an index rebuild, some customers who are making small changes choose to "version" a field by creating a new one that coexists alongside a previous version. Over time, this leads to orphaned content in the form of obsolete fields or obsolete custom analyzer definitions, especially in a production index that is expensive to replicate. You can address these issues on planned updates to the index as part of index lifecycle management.

Endpoint connection and security

All indexing and query requests target an index. Endpoints are usually one of the following:

Endpoint	Connection and access control
`<your-service>.search.windows.net/indexes`	Targets the indexes collection. Used when creating, listing, or deleting an index. Admin rights are required for these operations, available through admin API keys or a Search Contributor role.
`<your-service>.search.windows.net/indexes/<your-index>/docs`	Targets the documents collection of a single index. Used when querying an index or data refresh. For queries, read rights are sufficient, and available through query API keys or a data reader role. For data refresh, admin rights are required.

Next steps

You can get hands-on experience creating an index using almost any sample or walkthrough for Cognitive Search. For starters, you could choose any of the quickstarts from the table of contents.

But you'll also want to become familiar with methodologies for loading an index with data. Index definition and data import strategies are defined in tandem. The following articles provide more information about creating and loading an index.

- Create a search index
- Create an index alias
- Data import overview
- Add, Update or Delete Documents (REST)

Data import in Azure Cognitive Search

Article • 12/16/2022 • 4 minutes to read

In Azure Cognitive Search, queries execute over user-owned content that's loaded into a search index. This article describes the two basic workflows for populating an index: *push* your data into the index programmatically, or *pull* in the data using a search indexer.

With either approach, the objective is to load data from an external data source. Although you can create an empty index, it's not queryable until you add the content.

> ⓘ **Note**
>
> If **AI enrichment** is a solution requirement, you must use the pull model (indexers) to load an index. Skillsets are attached to an indexer and don't run independently.

Pushing data to an index

The push model, used to programmatically send your data to Azure Cognitive Search, is the most flexible approach for the following reasons:

- First, there are no restrictions on data source type. The dataset must be composed of JSON documents that map to your index schema, but the data can come from anywhere.
- Second, there are no restrictions on frequency of execution. You can push changes to an index as often as you like. For applications having low latency requirements (for example, if you need search operations to be in sync with dynamic inventory databases), the push model is your only option.
- Third, you can upload documents individually or in batches up to 1000 per batch, or 16 MB per batch, whichever limit comes first.
- Fourth, connectivity and the secure retrieval of documents are fully under your control. In contrast, indexer connections are authenticated using the security features provided in Cognitive Search.

How to push data to an Azure Cognitive Search index

You can use the following APIs to load single or multiple documents into an index:

- Add, Update, or Delete Documents (REST API)
- IndexDocumentsAction class (Azure SDK for .NET) or IndexDocumentsBatch class

There is currently no tool support for pushing data via the portal.

For an introduction to the push APIs, see:

- C# Tutorial: Optimize indexing with the push API
- C# Quickstart: Create an Azure Cognitive Search index using .NET SDK
- REST Quickstart: Create an Azure Cognitive Search index using PowerShell
- Python Quickstart: Create an Azure Cognitive Search index using the Azure SDK for Python
- JavaScript Quickstart: Create an Azure Cognitive Search index using the Azure SDK for JavaScript

Indexing actions: upload, merge, mergeOrUpload, delete

You can control the type of indexing action on a per-document basis, specifying whether the document should be uploaded in full, merged with existing document content, or deleted.

Whether you use the REST API or an SDK, the following document operations are supported for data import:

- **Upload**, similar to an "upsert" where the document is inserted if it is new, and updated or replaced if it exists. If the document is missing values that the index requires, the document field's value is set to null.

- **merge** updates a document that already exists, and fails a document that cannot be found. Merge replaces existing values. For this reason, be sure to check for collection fields that contain multiple values, such as fields of type `Collection(Edm.String)`. For example, if a `tags` field starts with a value of `["budget"]` and you execute a merge with `["economy", "pool"]`, the final value of the `tags` field is `["economy", "pool"]`. It won't be `["budget", "economy", "pool"]`.

- **mergeOrUpload** behaves like **merge** if the document exists, and **upload** if the document is new.

- **delete** removes the entire document from the index. If you want to remove an individual field, use **merge** instead, setting the field in question to null.

Pulling data into an index

The pull model crawls a supported data source and automatically uploads the data into your index. In Azure Cognitive Search, this capability is implemented through *indexers*, currently available for these platforms:

- Azure Blob storage
- Azure Table storage
- Azure Data Lake Storage Gen2
- Azure Files (preview)
- Azure Cosmos DB
- Azure SQL Database, SQL Managed Instance, and SQL Server on Azure VMs
- SharePoint in Microsoft 365 (preview)

Indexers connect an index to a data source (usually a table, view, or equivalent structure), and map source fields to equivalent fields in the index. During execution, the rowset is automatically transformed to JSON and loaded into the specified index. All indexers support schedules so that you can specify how frequently the data is to be refreshed. Most indexers provide change tracking if the data source supports it. By tracking changes and deletes to existing documents in addition to recognizing new documents, indexers remove the need to actively manage the data in your index.

How to pull data into an Azure Cognitive Search index

Indexer functionality is exposed in the Azure portal, the REST API, and the .NET SDK.

An advantage to using the portal is that Azure Cognitive Search can usually generate a default index schema by reading the metadata of the source dataset. You can modify the generated index until the index is processed, after which the only schema edits allowed are those that do not require reindexing. If the changes affect the schema itself, you would need to rebuild the index.

Verify data import with Search explorer

A quick way to perform a preliminary check on the document upload is to use **Search explorer** in the portal.

The explorer lets you query an index without having to write any code. The search experience is based on default settings, such as the simple syntax and default searchMode query parameter. Results are returned in JSON so that you can inspect the entire document.

Here is an example query that you can run in Search Explorer. The "HotelId" is the document key of the hotels-sample-index. The filter provides the document ID of a specific document:

```HTTP
$filter=HotelId eq '50'
```

If you're using REST, this Look up query achieves the same purpose.

See also

- Indexer overview
- Portal quickstart: create, load, query an index

Import data wizard in Azure Cognitive Search

Article • 10/21/2022 • 9 minutes to read

The **Import data wizard** in the Azure portal creates multiple objects used for indexing and AI enrichment on a search service. If you're new to Azure Cognitive Search, it's one of the most powerful features at your disposal. With minimal effort, you can create an indexing or enrichment pipeline that exercises most of the functionality of Azure Cognitive Search.

If you're using the wizard for proof-of-concept testing, this article explains the internal workings of the wizard so that you can use it more effectively.

This article isn't a step by step. For help with using the wizard with built-in sample data, see the Quickstart: Create a search index or Quickstart: Create a text translation and entity skillset.

Starting the wizard

In the Azure portal, open the search service page from the dashboard or find your service in the service list. In the service Overview page at the top, select **Import data**.

The wizard opens fully expanded in the browser window so that you have more room to work.

You can also launch **Import data** from other Azure services, including Azure Cosmos DB, Azure SQL Database, SQL Managed Instance, and Azure Blob Storage. Look for **Add Azure Cognitive Search** in the left-navigation pane on the service overview page.

Objects created by the wizard

The wizard will output the objects in the following table. After the objects are created, you can review their JSON definitions in the portal or call them from code.

Object	Description

Object	Description
Indexer	A configuration object specifying a data source, target index, an optional skillset, optional schedule, and optional configuration settings for error handing and base-64 encoding.
Data Source	Persists connection information to a supported data source on Azure. A data source object is used exclusively with indexers.
Index	Physical data structure used for full text search and other queries.
Skillset	Optional. A complete set of instructions for manipulating, transforming, and shaping content, including analyzing and extracting information from image files. Unless the volume of work fall under the limit of 20 transactions per indexer per day, the skillset must include a reference to a Cognitive Services resource that provides enrichment.
Knowledge store	Optional. Stores output from an AI enrichment pipeline in tables and blobs in Azure Storage for independent analysis or downstream processing.

Benefits and limitations

Before writing any code, you can use the wizard for prototyping and proof-of-concept testing. The wizard connects to external data sources, samples the data to create an initial index, and then imports the data as JSON documents into an index on Azure Cognitive Search.

If you're evaluating skillsets, the wizard will handle all of the output field mappings and add helper functions to create usable objects. Text split is added if you specify a parsing mode. Text merge is added if you chose image analysis so that the wizard can reunite text descriptions with image content. Shaper skills added to support valid projections if you chose the knowledge store option. All of the above tasks come with a learning curve. If you're new to enrichment, the ability to have these steps handled for you allows you to measure the value of a skill without having to invest much time and effort.

Sampling is the process by which an index schema is inferred, and it has some limitations. When the data source is created, the wizard picks a random sample of documents to decide what columns are part of the data source. Not all files are read, as this could potentially take hours for very large data sources. Given a selection of documents, source metadata, such as field name or type, is used to create a fields collection in an index schema. Depending on the complexity of source data, you might need to edit the initial schema for accuracy, or extend it for completeness. You can make your changes inline on the index definition page.

Overall, the advantages of using the wizard are clear: as long as requirements are met, you can prototype a queryable index within minutes. Some of the complexities of indexing, such as serializing data as JSON documents, are handled by the wizard.

The wizard isn't without limitations. Constraints are summarized as follows:

- The wizard doesn't support iteration or reuse. Each pass through the wizard creates a new index, skillset, and indexer configuration. Only data sources can be persisted and reused within the wizard. To edit or refine other objects, either delete the objects and start over, or use the REST APIs or .NET SDK to modify the structures.
- Source content must reside in a supported data source.
- Sampling is over a subset of source data. For large data sources, it's possible for the wizard to miss fields. You might need to extend the schema, or correct the inferred data types, if sampling is insufficient.
- AI enrichment, as exposed in the portal, is limited to a subset of built-in skills.
- A knowledge store, which can be created by the wizard, is limited to a few default projections and uses a default naming convention. If you want to customize names or projections, you'll need to create the knowledge store through REST API or the SDKs.
- Public access to all networks must be enabled on the supported data source while the wizard is used, since the portal won't be able to access the data source during setup if public access is disabled. This means that if your data source has a firewall enabled, you must disable it, run the Import Data wizard and then enable it after wizard setup is completed. If this isn't an option, you can create Azure Cognitive Search data source, indexer, skillset and index through REST API or the SDKs.

Workflow

The wizard is organized into four main steps:

1. Connect to a supported Azure data source.
2. Create an index schema, inferred by sampling source data.
3. Optionally, add AI enrichments to extract or generate content and structure. Inputs for creating a knowledge store are collected in this step.

4. Run the wizard to create objects, load data, set a schedule and other configuration options.

The workflow is a pipeline, so it's one way. You can't use the wizard to edit any of the objects that were created, but you can use other portal tools, such as the index or indexer designer or the JSON editors, for allowed updates.

Data source configuration in the wizard

The **Import data** wizard connects to an external supported data source using the internal logic provided by Azure Cognitive Search indexers, which are equipped to sample the source, read metadata, crack documents to read content and structure, and serialize contents as JSON for subsequent import to Azure Cognitive Search.

You can paste in a connection to a supported data source in a different subscription or region, but the **Choose an existing connection** picker is scoped to the active subscription.

Not all preview data sources are guaranteed to be available in the wizard. Because each data source has the potential for introducing other changes downstream, a preview data

source will only be added to the data sources list if it fully supports all of the experiences in the wizard, such as skillset definition and index schema inference.

You can only import from a single table, database view, or equivalent data structure, however the structure can include hierarchical or nested substructures. For more information, see How to model complex types.

Skillset configuration in the wizard

Skillset configuration occurs after the data source definition because the type of data source will inform the availability of certain built-in skills. In particular, if you're indexing files from Blob Storage, your choice of parsing mode of those files will determine whether sentiment analysis is available.

The wizard will add the skills you choose, but it will also add other skills that are necessary for achieving a successful outcome. For example, if you specify a knowledge store, the wizard adds a Shaper skill to support projections (or physical data structures).

Skillsets are optional and there's a button at the bottom of the page to skip ahead if you don't want AI enrichment.

Index schema configuration in the wizard

The wizard samples your data source to detect the fields and field type. Depending on the data source, it might also offer fields for indexing metadata.

Because sampling is an imprecise exercise, review the index for the following considerations:

1. Is the field list accurate? If your data source contains fields that weren't picked up in sampling, you can manually add any new fields that sampling missed, and remove any that don't add value to a search experience or that won't be used in a filter expression or scoring profile.

2. Is the data type appropriate for the incoming data? Azure Cognitive Search supports the entity data model (EDM) data types. For Azure SQL data, there's mapping chart that lays out equivalent values. For more background, see Field mappings and transformations.

3. Do you have one field that can serve as the *key*? This field must be Edm.string and it must uniquely identify a document. For relational data, it might be mapped to a primary key. For blobs, it might be the `metadata-storage-path`. If field values include spaces or dashes, you must set the **Base-64 Encode Key** option in the

Create an Indexer step, under **Advanced options**, to suppress the validation check for these characters.

4. Set attributes to determine how that field is used in an index.

   Take your time with this step because attributes determine the physical expression of fields in the index. If you want to change attributes later, even programmatically, you'll almost always need to drop and rebuild the index. Core attributes like **Searchable** and **Retrievable** have a negligible impact on storage. Enabling filters and using suggesters increase storage requirements.

   - **Searchable** enables full-text search. Every field used in free form queries or in query expressions must have this attribute. Inverted indexes are created for each field that you mark as **Searchable**.
   - **Retrievable** returns the field in search results. Every field that provides content to search results must have this attribute. Setting this field doesn't appreciably affect index size.
   - **Filterable** allows the field to be referenced in filter expressions. Every field used in a **$filter** expression must have this attribute. Filter expressions are for exact matches. Because text strings remain intact, more storage is required to accommodate the verbatim content.
   - **Facetable** enables the field for faceted navigation. Only fields also marked as **Filterable** can be marked as **Facetable**.
   - **Sortable** allows the field to be used in a sort. Every field used in an **$Orderby** expression must have this attribute.

5. Do you need lexical analysis? For Edm.string fields that are **Searchable**, you can set an **Analyzer** if you want language-enhanced indexing and querying.

   The default is *Standard Lucene* but you could choose *Microsoft English* if you wanted to use Microsoft's analyzer for advanced lexical processing, such as resolving irregular noun and verb forms. Only language analyzers can be specified in the portal. Using a custom analyzer or a non-language analyzer like Keyword, Pattern, and so forth, must be done programmatically. For more information about analyzers, see Add language analyzers.

6. Do you need typeahead functionality in the form of autocomplete or suggested results? Select the **Suggester** the checkbox to enable typeahead query suggestions and autocomplete on selected fields. Suggesters add to the number of tokenized terms in your index, and thus consume more storage.

Indexer configuration in the wizard

The last page of the wizard collects user inputs for indexer configuration. You can specify a schedule and set other options that will vary by the data source type.

Internally, the wizard also sets up the following definitions, which aren't visible in the indexer until after it's created:

- field mappings between the data source and index
- output field mappings between skill output and an index

Next steps

The best way to understand the benefits and limitations of the wizard is to step through it. The following quickstart explains each step.

Quickstart: Create a search index using the Azure portal

Analyzers for text processing in Azure Cognitive Search

Article • 06/24/2022 • 10 minutes to read

An *analyzer* is a component of the full text search engine that's responsible for processing strings during indexing and query execution. Text processing (also known as lexical analysis) is transformative, modifying a string through actions such as these:

- Remove non-essential words (stopwords) and punctuation
- Split up phrases and hyphenated words into component parts
- Lower-case any upper-case words
- Reduce words into primitive root forms for storage efficiency and so that matches can be found regardless of tense

Analysis applies to `Edm.String` fields that are marked as "searchable", which indicates full text search.

For fields of this configuration, analysis occurs during indexing when tokens are created, and then again during query execution when queries are parsed and the engine scans for matching tokens. A match is more likely to occur when the same analyzer is used for both indexing and queries, but you can set the analyzer for each workload independently, depending on your requirements.

Query types that are *not* full text search, such as filters or fuzzy search, don't go through the analysis phase on the query side. Instead, the parser sends those strings directly to the search engine, using the pattern that you provide as the basis for the match. Typically, these query forms require whole-string tokens to make pattern matching work. To ensure whole terms tokens during indexing, you might need custom analyzers. For more information about when and why query terms are analyzed, see Full text search in Azure Cognitive Search.

For more background on lexical analysis, listen to the following video clip for a brief explanation.
https://www.youtube-nocookie.com/embed/Y_X6USgvB1g?version=3&start=132&end=189

Default analyzer

In Azure Cognitive Search, an analyzer is automatically invoked on all string fields marked as searchable.

By default, Azure Cognitive Search uses the Apache Lucene Standard analyzer (standard lucene), which breaks text into elements following the "Unicode Text Segmentation" rules. The standard analyzer converts all characters to their lower case form. Both indexed documents and search terms go through the analysis during indexing and query processing.

You can override the default on a field-by-field basis. Alternative analyzers can be a language analyzer for linguistic processing, a custom analyzer, or a built-in analyzer from the list of available analyzers.

Types of analyzers

The following list describes which analyzers are available in Azure Cognitive Search.

Category	Description
Standard Lucene analyzer	Default. No specification or configuration is required. This general-purpose analyzer performs well for many languages and scenarios.
Built-in analyzers	Consumed as-is and referenced by name. There are two types: language and language-agnostic. Specialized (language-agnostic) analyzers are used when text inputs require specialized processing or minimal processing. Examples of analyzers in this category include **Asciifolding**, **Keyword**, **Pattern**, **Simple**, **Stop**, **Whitespace**. Language analyzers are used when you need rich linguistic support for individual languages. Azure Cognitive Search supports 35 Lucene language analyzers and 50 Microsoft natural language processing analyzers.
Custom analyzers	Refers to a user-defined configuration of a combination of existing elements, consisting of one tokenizer (required) and optional filters (char or token).

A few built-in analyzers, such as **Pattern** or **Stop**, support a limited set of configuration options. To set these options, create a custom analyzer, consisting of the built-in analyzer and one of the alternative options documented in Built-in analyzers. As with any custom configuration, provide your new configuration with a name, such as *myPatternAnalyzer* to distinguish it from the Lucene Pattern analyzer.

Specifying analyzers

Setting an analyzer is optional. As a general rule, try using the default standard Lucene analyzer first to see how it performs. If queries fail to return the expected results,

switching to a different analyzer is often the right solution.

1. If you're using a custom analyzer, add it to the search index under the "analyzer" section. For more information, see Create Index and also Add custom analyzers.

2. When defining a field, set it's "analyzer" property to one of the following: a built-in analyzer such as **keyword**, a language analyzer such as `en.microsoft`, or a custom analyzer (defined in the same index schema).

   JSON

   ```
    "fields": [
   {
     "name": "Description",
     "type": "Edm.String",
     "retrievable": true,
     "searchable": true,
     "analyzer": "en.microsoft",
     "indexAnalyzer": null,
     "searchAnalyzer": null
   },
   ```

3. If you're using a language analyzer, you must use the "analyzer" property to specify it. The "searchAnalyzer" and "indexAnalyzer" properties don't apply to language analyzers.

4. Alternatively, set "indexAnalyzer" and "searchAnalyzer" to vary the analyzer for each workload. These properties work together as a substitute for the "analyzer" property, which must be null. You might use different analyzers for indexing and queries if one of those activities required a specific transformation not needed by the other.

   JSON

   ```
    "fields": [
   {
     "name": "ProductGroup",
     "type": "Edm.String",
     "retrievable": true,
     "searchable": true,
     "analyzer": null,
     "indexAnalyzer": "keyword",
     "searchAnalyzer": "standard"
   },
   ```

When to add analyzers

The best time to add and assign analyzers is during active development, when dropping and recreating indexes is routine.

Because analyzers are used to tokenize terms, you should assign an analyzer when the field is created. In fact, assigning an analyzer or indexAnalyzer to a field that has already been physically created isn't allowed (although you can change the searchAnalyzer property at any time with no impact to the index).

To change the analyzer of an existing field, you'll have to drop and recreate the entire index (you can't rebuild individual fields). For indexes in production, you can defer a rebuild by creating a new field with the new analyzer assignment, and start using it in place of the old one. Use Update Index to incorporate the new field and mergeOrUpload to populate it. Later, as part of planned index servicing, you can clean up the index to remove obsolete fields.

To add a new field to an existing index, call Update Index to add the field, and mergeOrUpload to populate it.

To add a custom analyzer to an existing index, pass the "allowIndexDowntime" flag in Update Index if you want to avoid this error:

```
"Index update not allowed because it would cause downtime. In order to add new analyzers, tokenizers, token filters, or character filters to an existing index, set the 'allowIndexDowntime' query parameter to 'true' in the index update request. Note that this operation will put your index offline for at least a few seconds, causing your indexing and query requests to fail. Performance and write availability of the index can be impaired for several minutes after the index is updated, or longer for very large indexes."
```

Recommendations for working with analyzers

This section offers advice on how to work with analyzers.

One analyzer for read-write unless you have specific requirements

Azure Cognitive Search lets you specify different analyzers for indexing and search through the "indexAnalyzer" and "searchAnalyzer" field properties. If unspecified, the analyzer set with the analyzer property is used for both indexing and searching. If the analyzer is unspecified, the default Standard Lucene analyzer is used.

A general rule is to use the same analyzer for both indexing and querying, unless specific requirements dictate otherwise. Be sure to test thoroughly. When text processing differs at search and indexing time, you run the risk of mismatch between query terms and indexed terms when the search and indexing analyzer configurations aren't aligned.

Test during active development

Overriding the standard analyzer requires an index rebuild. If possible, decide on which analyzers to use during active development, before rolling an index into production.

Inspect tokenized terms

If a search fails to return expected results, the most likely scenario is token discrepancies between term inputs on the query, and tokenized terms in the index. If the tokens aren't the same, matches fail to materialize. To inspect tokenizer output, we recommend using the Analyze API as an investigation tool. The response consists of tokens, as generated by a specific analyzer.

REST examples

The examples below show analyzer definitions for a few key scenarios.

- Custom analyzer example
- Assign analyzers to a field example
- Mixing analyzers for indexing and search
- Language analyzer example

Custom analyzer example

This example illustrates an analyzer definition with custom options. Custom options for char filters, tokenizers, and token filters are specified separately as named constructs, and then referenced in the analyzer definition. Predefined elements are used as-is and referenced by name.

Walking through this example:

- Analyzers are a property of the field class for a searchable field.

- A custom analyzer is part of an index definition. It might be lightly customized (for example, customizing a single option in one filter) or customized in multiple

places.

- In this case, the custom analyzer is "my_analyzer", which in turn uses a customized standard tokenizer "my_standard_tokenizer" and two token filters: lowercase and customized asciifolding filter "my_asciifolding".
- It also defines 2 custom char filters "map_dash" and "remove_whitespace". The first one replaces all dashes with underscores while the second one removes all spaces. Spaces need to be UTF-8 encoded in the mapping rules. The char filters are applied before tokenization and will affect the resulting tokens (the standard tokenizer breaks on dash and spaces but not on underscore).

JSON

```
{
   "name":"myindex",
   "fields":[
      {
         "name":"id",
         "type":"Edm.String",
         "key":true,
         "searchable":false
      },
      {
         "name":"text",
         "type":"Edm.String",
         "searchable":true,
         "analyzer":"my_analyzer"
      }
   ],
   "analyzers":[
      {
         "name":"my_analyzer",
         "@odata.type":"#Microsoft.Azure.Search.CustomAnalyzer",
         "charFilters":[
            "map_dash",
            "remove_whitespace"
         ],
         "tokenizer":"my_standard_tokenizer",
         "tokenFilters":[
            "my_asciifolding",
            "lowercase"
         ]
      }
   ],
   "charFilters":[
      {
         "name":"map_dash",
         "@odata.type":"#Microsoft.Azure.Search.MappingCharFilter",
         "mappings":["-=>_"]
      },
      {
```

```
            "name":"remove_whitespace",
            "@odata.type":"#Microsoft.Azure.Search.MappingCharFilter",
            "mappings":["\\u0020=>"]
         }
      ],
      "tokenizers":[
         {
            "name":"my_standard_tokenizer",
            "@odata.type":"#Microsoft.Azure.Search.StandardTokenizerV2",
            "maxTokenLength":20
         }
      ],
      "tokenFilters":[
         {
            "name":"my_asciifolding",
            "@odata.type":"#Microsoft.Azure.Search.AsciiFoldingTokenFilter",
            "preserveOriginal":true
         }
      ]
   }
```

Per-field analyzer assignment example

The Standard analyzer is the default. Suppose you want to replace the default with a different predefined analyzer, such as the pattern analyzer. If you aren't setting custom options, you only need to specify it by name in the field definition.

The "analyzer" element overrides the Standard analyzer on a field-by-field basis. There is no global override. In this example, `text1` uses the pattern analyzer and `text2`, which doesn't specify an analyzer, uses the default.

JSON

```
   {
      "name":"myindex",
      "fields":[
         {
            "name":"id",
            "type":"Edm.String",
            "key":true,
            "searchable":false
         },
         {
            "name":"text1",
            "type":"Edm.String",
            "searchable":true,
            "analyzer":"pattern"
         },
         {
            "name":"text2",
```

```
            "type":"Edm.String",
            "searchable":true
         }
      ]
   }
```

Mixing analyzers for indexing and search operations

The APIs include index attributes for specifying different analyzers for indexing and search. The searchAnalyzer and indexAnalyzer attributes must be specified as a pair, replacing the single analyzer attribute.

JSON

```
   {
      "name":"myindex",
      "fields":[
         {
            "name":"id",
            "type":"Edm.String",
            "key":true,
            "searchable":false
         },
         {
            "name":"text",
            "type":"Edm.String",
            "searchable":true,
            "indexAnalyzer":"whitespace",
            "searchAnalyzer":"simple"
         },
      ],
   }
```

Language analyzer example

Fields containing strings in different languages can use a language analyzer, while other fields retain the default (or use some other predefined or custom analyzer). If you use a language analyzer, it must be used for both indexing and search operations. Fields that use a language analyzer can't have different analyzers for indexing and search.

JSON

```
   {
      "name":"myindex",
      "fields":[
         {
            "name":"id",
```

```
            "type":"Edm.String",
            "key":true,
            "searchable":false
        },
        {
            "name":"text",
            "type":"Edm.String",
            "searchable":true,
            "indexAnalyzer":"whitespace",
            "searchAnalyzer":"simple"
        },
        {
            "name":"text_fr",
            "type":"Edm.String",
            "searchable":true,
            "analyzer":"fr.lucene"
        }
    ],
}
```

C# examples

If you are using the .NET SDK code samples, you can append these examples to use or configure analyzers.

- Assign a built-in analyzer
- Configure an analyzer

Assign a language analyzer

Any analyzer that is used as-is, with no configuration, is specified on a field definition. There is no requirement for creating an entry in the **[analyzers]** section of the index.

Language analyzers are used as-is. To use them, call LexicalAnalyzer, specifying the LexicalAnalyzerName type providing a text analyzer supported in Azure Cognitive Search.

Custom analyzers are similarly specified on the field definition, but for this to work you must specify the analyzer in the index definition, as described in the next section.

C#

```
    public partial class Hotel
    {
        . . .
        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.EnLucene)]
```

```
        public string Description { get; set; }

        [SearchableField(AnalyzerName =
LexicalAnalyzerName.Values.FrLucene)]
        [JsonPropertyName("Description_fr")]
        public string DescriptionFr { get; set; }

        [SearchableField(AnalyzerName = "url-analyze")]
        public string Url { get; set; }
      . . .
    }
```

Define a custom analyzer

When customization or configuration is required, add an analyzer construct to an index. Once you define it, you can add it the field definition as demonstrated in the previous example.

Create a CustomAnalyzer object. A custom analyzer is a user-defined combination of a known tokenizer, zero or more token filter, and zero or more character filter names:

- CustomAnalyzer.Tokenizer
- CustomAnalyzer.TokenFilters
- CustomAnalyzer.CharFilters

The following example creates a custom analyzer named "url-analyze" that uses the uax_url_email tokenizer and the Lowercase token filter.

C#

```
private static void CreateIndex(string indexName, SearchIndexClient
adminClient)
{
   FieldBuilder fieldBuilder = new FieldBuilder();
   var searchFields = fieldBuilder.Build(typeof(Hotel));

   var analyzer = new CustomAnalyzer("url-analyze", "uax_url_email")
   {
         TokenFilters = { TokenFilterName.Lowercase }
   };

   var definition = new SearchIndex(indexName, searchFields);

   definition.Analyzers.Add(analyzer);

   adminClient.CreateOrUpdateIndex(definition);
}
```

For more examples, see CustomAnalyzerTests.cs.

Next steps

A detailed description of query execution can be found in Full text search in Azure Cognitive Search. The article uses examples to explain behaviors that might seem counter-intuitive on the surface.

To learn more about analyzers, see the following articles:

- Add a language analyzer
- Add a custom analyzer
- Create a search index
- Create a multi-language index

Indexers in Azure Cognitive Search

Article • 02/24/2023 • 8 minutes to read

An *indexer* in Azure Cognitive Search is a crawler that extracts searchable content from cloud data sources and populates a search index using field-to-field mappings between source data and a search index. This approach is sometimes referred to as a 'pull model' because the search service pulls data in without you having to write any code that adds data to an index. Indexers also drive the AI enrichment capabilities of Cognitive Search, integrating external processing of content en route to an index.

Indexers are cloud-only, with individual indexers for supported data sources. When configuring an indexer, you'll specify a data source (origin) and a search index (destination). Several sources, such as Azure Blob Storage, have more configuration properties specific to that content type.

You can run indexers on demand or on a recurring data refresh schedule that runs as often as every five minutes. More frequent updates require a 'push model' that simultaneously updates data in both Azure Cognitive Search and your external data source.

Indexer scenarios and use cases

You can use an indexer as the sole means for data ingestion, or in combination with other techniques. The following table summarizes the main scenarios.

Scenario	Strategy
Single data source	This pattern is the simplest: one data source is the sole content provider for a search index. Most supported data sources provide some form of change detection so that subsequent indexer runs pick up the difference when content is added or updated in the source.
Multiple data sources	An indexer specification can have only one data source, but the search index itself can accept content from multiple sources, where each indexer run brings new content from a different data provider. Each source can contribute its share of full documents, or populate selected fields in each document. For a closer look at this scenario, see Tutorial: Index from multiple data sources.

Scenario	Strategy
Multiple indexers	Multiple data sources are typically paired with multiple indexers if you need to vary run time parameters, the schedule, or field mappings. Cross-region scale out of Cognitive Search is another scenario. You might have copies of the same search index in different regions. To synchronize search index content, you could have multiple indexers pulling from the same data source, where each indexer targets a different search index in each region. Parallel indexing of very large data sets also requires a multi-indexer strategy, where each indexer targets a subset of the data.
Content transformation	Indexers drive AI enrichment. Content transforms are defined in a skillset that you attach to the indexer.

Supported data sources

Indexers crawl data stores on Azure and outside of Azure.

- Azure Blob Storage
- Azure Cosmos DB
- Azure Data Lake Storage Gen2
- Azure SQL Database
- Azure Table Storage
- Azure SQL Managed Instance
- SQL Server on Azure Virtual Machines
- Azure Files (in preview)
- Azure MySQL (in preview)
- SharePoint in Microsoft 365 (in preview)
- Azure Cosmos DB for MongoDB (in preview)
- Azure Cosmos DB for Apache Gremlin (in preview)

Indexers accept flattened row sets, such as a table or view, or items in a container or folder. In most cases, it creates one search document per row, record, or item.

Indexer connections to remote data sources can be made using standard Internet connections (public) or encrypted private connections when you use Azure virtual networks for client apps. You can also set up connections to authenticate using a managed identity. For more information about secure connections, see Indexer access to content protected by Azure network security features and Connect to a data source using a managed identity.

Stages of indexing

On an initial run, when the index is empty, an indexer will read in all of the data provided in the table or container. On subsequent runs, the indexer can usually detect and retrieve just the data that has changed. For blob data, change detection is automatic. For other data sources like Azure SQL or Azure Cosmos DB, change detection must be enabled.

For each document it receives, an indexer implements or coordinates multiple steps, from document retrieval to a final search engine "handoff" for indexing. Optionally, an indexer also drives skillset execution and outputs, assuming a skillset is defined.

Stage 1: Document cracking

Document cracking is the process of opening files and extracting content. Text-based content can be extracted from files on a service, rows in a table, or items in container or collection. If you add a skillset and image skills, document cracking can also extract images and queue them for image processing.

Depending on the data source, the indexer will try different operations to extract potentially indexable content:

- When the document is a file with embedded images, such as a PDF, the indexer extracts text, images, and metadata. Indexers can open files from Azure Blob Storage, Azure Data Lake Storage Gen2, and SharePoint.
- When the document is a record in Azure SQL, the indexer will extract non-binary content from each field in each record.
- When the document is a record in Azure Cosmos DB, the indexer will extract non-binary content from fields and subfields from the Azure Cosmos DB document.

Stage 2: Field mappings

An indexer extracts text from a source field and sends it to a destination field in an index or knowledge store. When field names and data types coincide, the path is clear. However, you might want different names or types in the output, in which case you need to tell the indexer how to map the field.

To specify field mappings, enter the source and destination fields in the indexer definition.

Field mapping occurs after document cracking, but before transformations, when the indexer is reading from the source documents. When you define a field mapping, the value of the source field is sent as-is to the destination field with no modifications.

Stage 3: Skillset execution

Skillset execution is an optional step that invokes built-in or custom AI processing. Skillsets can add optical character recognition (OCR) or other forms of image analysis if the content is binary. Skillsets can also add natural language processing. For example, you can add text translation or key phrase extraction.

Whatever the transformation, skillset execution is where enrichment occurs. If an indexer is a pipeline, you can think of a skillset as a "pipeline within the pipeline".

Stage 4: Output field mappings

If you include a skillset, you'll need to specify output field mappings in the indexer definition. The output of a skillset is manifested internally as a tree structure referred to as an *enriched document*. Output field mappings allow you to select which parts of this tree to map into fields in your index.

Despite the similarity in names, output field mappings and field mappings build associations from different sources. Field mappings associate the content of source field to a destination field in a search index. Output field mappings associate the content of an internal enriched document (skill outputs) to destination fields in the index. Unlike field mappings, which are considered optional, an output field mapping is required for any transformed content that should be in the index.

The next image shows a sample indexer debug session representation of the indexer stages: document cracking, field mappings, skillset execution, and output field mappings.

Basic workflow

Indexers can offer features that are unique to the data source. In this respect, some aspects of indexer or data source configuration will vary by indexer type. However, all indexers share the same basic composition and requirements. Steps that are common to all indexers are covered below.

Step 1: Create a data source

Indexers require a *data source* object that provides a connection string and possibly credentials. Data sources are independent objects. Multiple indexers can use the same data source object to load more than one index at a time.

You can create a data source using any of these approaches:

- Using the Azure portal, on the **Data sources** tab of your search service pages, select **Add data source** to specify the data source definition.
- Using the Azure portal, the Import data wizard outputs a data source.
- Using the REST APIs, call Create Data Source.
- Using the Azure SDK for .NET, call SearchIndexerDataSourceConnection class

Step 2: Create an index

An indexer will automate some tasks related to data ingestion, but creating an index is generally not one of them. As a prerequisite, you must have a predefined index that contains corresponding target fields for any source fields in your external data source. Fields need to match by name and data type. If not, you can define field mappings to establish the association.

For more information, see Create an index.

Step 3: Create and run (or schedule) the indexer

An indexer definition consists of properties that uniquely identify the indexer, specify which data source and index to use, and provide other configuration options that influence run time behaviors, including whether the indexer runs on demand or on a schedule.

Any errors or warnings about data access or skillset validation will occur during indexer execution. Until indexer execution starts, dependent objects such as data sources, indexes, and skillsets are passive on the search service.

For more information, see Create an indexer

After the first indexer run, you can rerun it on demand or set up a schedule.

You can monitor indexer status in the portal or through Get Indexer Status API. You should also run queries on the index to verify the result is what you expected.

Indexers don't have dedicated processing resources. Based on this, indexers' status may show as idle before running (depending on other jobs in the queue) and run times may

not be predictable. Other factors define indexer performance as well, such as document size, document complexity, image analysis, among others.

Next steps

Now that you've been introduced to indexers, a next step is to review indexer properties and parameters, scheduling, and indexer monitoring. Alternatively, you could return to the list of supported data sources for more information about a specific source.

- Create indexers
- Reset and run indexers
- Schedule indexers
- Define field mappings
- Monitor indexer status

AI enrichment in Azure Cognitive Search

Article • 10/12/2022 • 7 minutes to read

AI enrichment is the application of machine learning models over content that isn't full text searchable in its raw form. Through enrichment, analysis and inference are used to create searchable content and structure where none previously existed.

Because Azure Cognitive Search is a full text search solution, the purpose of AI enrichment is to improve the utility of your content in search-related scenarios:

- Translation and language detection for multi-lingual search
- Entity recognition extracts people, places, and other entities from large chunks of text
- Key phrase extraction identifies and then outputs important terms
- Optical Character Recognition (OCR) recognizes printed and handwritten text in binary files
- Image analysis describes image content and outputs the descriptions as searchable text fields

AI enrichment is an extension of an **indexer pipeline**. An enrichment pipeline has all of the components of an indexer pipeline (indexer, data source, index), plus a **skillset** that specifies atomic enrichment steps.

The following diagram shows the progression of AI enrichment:

Import is the first step. Here, the indexer connects to a data source and pulls content (documents) into the search service. Azure Blob Storage is the most common resource used in AI enrichment scenarios, but any supported data source can provide content.

Enrich & Index covers most of the AI enrichment pipeline:

- Enrichment starts when the indexer "cracks documents" and extracts images and text. The kind of processing that occurs next will depend on your data and which skills you've added to a skillset. If you have images, they can be forwarded to skills that perform image processing. Text content is queued for text and natural language processing. Internally, skills create an "enriched document" that collects the transformations as they occur.

- Enriched content is generated during skillset execution, and is temporary unless you save it. You can enable an enrichment cache to persist cracked documents and skill outputs for subsequent reuse during future skillset executions.

- To get content into a search index, the indexer must have mapping information for sending enriched content to target field. Field mappings (explicit or implicit) set the data path from source data to a search index. Output field mappings set the data path from enriched documents to an index.

- Indexing is the process wherein raw and enriched content is ingested into the physical data structures of a search index (its files and folders). Lexical analysis and tokenization occur in this step.

Exploration is the last step. Output is always a search index that you can query from a client app. Output can optionally be a knowledge store consisting of blobs and tables in Azure Storage that are accessed through data exploration tools or downstream processes. If you're creating a knowledge store, projections determine the data path for enriched content. The same enriched content can appear in both indexes and knowledge stores.

When to use AI enrichment

Enrichment is useful if raw content is unstructured text, image content, or content that needs language detection and translation. Applying AI through the **built-in skills** can unlock this content for full text search and data science applications.

You can also create **custom skills** to provide external processing. Open-source, third-party, or first-party code can be integrated into the pipeline as a custom skill. Classification models that identify salient characteristics of various document types fall into this category, but any external package that adds value to your content could be used.

Use-cases for built-in skills

Built-in skills are based on the Cognitive Services APIs: Computer Vision and Language Service. Unless your content input is small, expect to attach a billable Cognitive Services resource to run larger workloads.

A skillset that's assembled using built-in skills is well suited for the following application scenarios:

- **Image processing** skills include Optical Character Recognition (OCR) and identification of visual features, such as facial detection, image interpretation, image recognition (famous people and landmarks), or attributes like image orientation. These skills create text representations of image content for full text search in Azure Cognitive Search.
- **Machine translation** is provided by the Text Translation skill, often paired with language detection for multi-language solutions.
- **Natural language processing** analyzes chunks of text. Skills in this category include Entity Recognition, Sentiment Detection (including opinion mining), and Personal Identifiable Information Detection. With these skills, unstructured text is mapped as searchable and filterable fields in an index.

Use-cases for custom skills

Custom skills execute external code that you provide. Custom skills can support more complex scenarios, such as recognizing forms, or custom entity detection using a model that you provide and wrap in the custom skill web interface. Several examples of custom skills include:

- Forms Recognizer
- Bing Entity Search API
- Custom entity recognition

Custom skills aren't always complex. For example, if you have an existing package that provides pattern matching or a document classification model, you can wrap it in a custom skill.

Storing output

In Azure Cognitive Search, an indexer saves the output it creates. A single indexer run can create up to three data structures that contain enriched and indexed output.

Data store	Required	Location	Description

Data store	Required	Location	Description
searchable index	Required	Search service	Used for full text search and other query forms. Specifying an index is an indexer requirement. Index content is populated from skill outputs, plus any source fields that are mapped directly to fields in the index.
knowledge store	Optional	Azure Storage	Used for downstream apps like knowledge mining or data science. A knowledge store is defined within a skillset. Its definition determines whether your enriched documents are projected as tables or objects (files or blobs) in Azure Storage.
enrichment cache	Optional	Azure Storage	Used for caching enrichments for reuse in subsequent skillset executions. The cache stores imported, unprocessed content (cracked documents). It also stores the enriched documents created during skillset execution. Caching is particularly helpful if you're using image analysis or OCR, and you want to avoid the time and expense of reprocessing image files.

Indexes and knowledge stores are fully independent of each other. While you must attach an index to satisfy indexer requirements, if your sole objective is a knowledge store, you can ignore the index after it's populated. Avoid deleting it though. If you want to rerun the indexer and skillset, you'll need the index in order for the indexer to run.

Exploring content

After you've defined and loaded a search index or a knowledge store, you can explore its data.

Query a search index

Run queries to access the enriched content generated by the pipeline. The index is like any other you might create for Azure Cognitive Search: you can supplement text analysis with custom analyzers, invoke fuzzy search queries, add filters, or experiment with scoring profiles to tune search relevance.

Use data exploration tools on a knowledge store

In Azure Storage, a knowledge store can assume the following forms: a blob container of JSON documents, a blob container of image objects, or tables in Table Storage. You can use Storage Explorer, Power BI, or any app that connects to Azure Storage to access your content.

- A blob container captures enriched documents in their entirety, which is useful if you're creating a feed into other processes.
- A table is useful if you need slices of enriched documents, or if you want to include or exclude specific parts of the output. For analysis in Power BI, tables are the recommended data source for data exploration and visualization in Power BI.

Availability and pricing

Enrichment is available in regions that have Azure Cognitive Services. You can check the availability of enrichment on the Azure products available by region page. Enrichment is available in all regions except:

- Australia Southeast
- China North 2
- Germany West Central

Billing follows a pay-as-you-go pricing model. The costs of using built-in skills are passed on when a multi-region Cognitive Services key is specified in the skillset. There are also costs associated with image extraction, as metered by Cognitive Search. Text extraction and utility skills, however, aren't billable. For more information, see How you're charged for Azure Cognitive Search.

Checklist: A typical workflow

An enrichment pipeline consists of *indexers* that have *skillsets*. Post-indexing, you can query an index to validate your results.

Start with a subset of data in a supported data source. Indexer and skillset design is an iterative process. The work goes faster with a small representative data set.

1. Create a data source that specifies a connection to your data.
2. Create a skillset. Unless your project is small, you'll want to attach a Cognitive Services resource. If you're creating a knowledge store, define it within the skillset.
3. Create an index schema that defines a search index.
4. Create and run the indexer to bring all of the above components together. This step retrieves the data, runs the skillset, and loads the index.

   An indexer is also where you specify field mappings and output field mappings that set up the data path to a search index.

Optionally, enable enrichment caching in the indexer configuration. This step allows you to reuse existing enrichments later on.

5. Run queries to evaluate results or start a debug session to work through any skillset issues.

To repeat any of the above steps, reset the indexer before you run it. Or, delete and recreate the objects on each run (recommended if you're using the free tier). If you enabled caching the indexer will pull from the cache if data is unchanged at the source, and if your edits to the pipeline don't invalidate the cache.

Next steps

- Quickstart: Create a skillset for AI enrichment
- Tutorial: Learn about the AI enrichment REST APIs
- Skillset concepts
- Knowledge store concepts
- Create a skillset
- Create a knowledge store

Incremental enrichment and caching in Azure Cognitive Search

Article • 04/21/2023

> **Important**
>
> This feature is in public preview under **supplemental terms of use**. The **preview REST API** supports this feature.

Incremental enrichment refers to the use of cached enrichments during skillset execution so that only new and changed skills and documents incur AI processing. The cache contains the output from document cracking, plus the outputs of each skill for every document. Although caching is billable (it uses Azure Storage), the overall cost of enrichment is reduced because the costs of storage are less than image extraction and AI processing.

When you enable caching, the indexer evaluates your updates to determine whether existing enrichments can be pulled from the cache. Image and text content from the document cracking phase, plus skill outputs that are upstream or orthogonal to your edits, are likely to be reusable.

After performing the incremental enrichments as indicated by the skillset update, refreshed results are written back to the cache, and also to the search index or knowledge store.

Cache configuration

Physically, the cache is stored in a blob container in your Azure Storage account, one per indexer. Each indexer is assigned a unique and immutable cache identifier that corresponds to the container it is using.

The cache is created when you specify the "cache" property and run the indexer. Only enriched content can be cached. If your indexer does not have an attached skillset, then caching does not apply.

The following example illustrates an indexer with caching enabled. See Enable enrichment caching for full instructions. Notice that when adding the cache property, use preview API version, 2020-06-30-Preview or later, on the request.

JSON

```
POST https://[search service name].search.windows.net/indexers?api-version=2020-06-30-Preview
    {
        "name": "myIndexerName",
        "targetIndexName": "myIndex",
        "dataSourceName": "myDatasource",
        "skillsetName": "mySkillset",
        "cache" : {
            "storageConnectionString" : "<Your storage account connection string>",
            "enableReprocessing": true
        },
        "fieldMappings" : [],
        "outputFieldMappings": [],
        "parameters": []
    }
```

Cache management

The lifecycle of the cache is managed by the indexer. If an indexer is deleted, its cache is also deleted. If the "cache" property on the indexer is set to null or the connection string is changed, the existing cache is deleted on the next indexer run.

While incremental enrichment is designed to detect and respond to changes with no intervention on your part, there are parameters you can use to invoke specific behaviors:

- Prioritize new documents
- Bypass skillset checks
- Bypass data source checks
- Force skillset evaluation

Prioritize new documents

The "cache" property includes an "enableReprocessing" parameter. It's used to control processing over incoming documents already represented in the cache. When true (default), documents already in the cache are reprocessed when you rerun the indexer, assuming your skill update affects that doc.

When false, existing documents are not reprocessed, effectively prioritizing new, incoming content over existing content. You should only set "enableReprocessing" to false on a temporary basis. Having "enableReprocessing" set to true most of the time ensures that all documents, both new and existing, are valid per the current skillset definition.

Bypass skillset evaluation

Modifying a skill and reprocessing of that skill typically go hand in hand. However, some changes to a skill should not result in reprocessing (for example, deploying a custom skill to a new location or with a new access key). Most likely, these are peripheral modifications that have no genuine impact on the substance of the skill output itself.

If you know that a change to the skill is indeed superficial, you should override skill evaluation by setting the "disableCacheReprocessingChangeDetection" parameter to true:

1. Call Update Skillset and modify the skillset definition.
2. Append the "disableCacheReprocessingChangeDetection=true" parameter on the request.
3. Submit the change.

Setting this parameter ensures that only updates to the skillset definition are committed and the change isn't evaluated for effects on the existing cache. Use a preview API version, 2020-06-30-Preview or later.

HTTP

```
PUT https://[servicename].search.windows.net/skillsets/[skillset name]?api-version=2020-06-30-Preview&disableCacheReprocessingChangeDetection
```

Bypass data source validation checks

Most changes to a data source definition will invalidate the cache. However, for scenarios where you know that a change should not invalidate the cache - such as changing a connection string or rotating the key on the storage account - append the "ignoreResetRequirement" parameter on the data source update. Setting this parameter to true allows the commit to go through, without triggering a reset condition that would result in all objects being rebuilt and populated from scratch.

HTTP

```
PUT https://[search service].search.windows.net/datasources/[data source name]?api-version=2020-06-30-Preview&ignoreResetRequirement
```

Force skillset evaluation

The purpose of the cache is to avoid unnecessary processing, but suppose you make a change to a skill that the indexer doesn't detect (for example, changing something in external code, such as a custom skill).

In this case, you can use the Reset Skills to force reprocessing of a particular skill, including any downstream skills that have a dependency on that skill's output. This API accepts a POST request with a list of skills that should be invalidated and marked for reprocessing. After Reset Skills, follow with a Run Indexer request to invoke the pipeline processing.

Re-cache specific documents

Resetting an indexer will result in all documents in the search corpus being reprocessed. In scenarios where only a few documents need to be reprocessed, use Reset Documents (preview) to force reprocessing of specific documents. When a document is reset, the indexer invalidates the cache for that document, which is then reprocessed by reading it from the data source. For more information, see Run or reset indexers, skills, and documents.

To reset specific documents, the request provides a list of document keys as read from the search index. If the key is mapped to a field in the external data source, the value that you provide should be the one used in the search index.

Depending on how you call the API, the request will either append, overwrite, or queue up the key list:

- Calling the API multiple times with different keys appends the new keys to the list of document keys reset.
- Calling the API with the "overwrite" query string parameter set to true will overwrite the current list of document keys to be reset with the request's payload.
- Calling the API only results in the document keys being added to the queue of work the indexer performs. When the indexer is next invoked, either as scheduled or on demand, it will prioritize processing the reset document keys before any other changes from the data source.

The following example illustrates a reset document request:

HTTP

```
POST https://[search service name].search.windows.net/indexers/[indexer
name]/resetdocs?api-version=2020-06-30-Preview
    {
```

```
        "documentKeys" : [
            "key1",
            "key2",
            "key3"
        ]
    }
```

Changes that invalidate the cache

Once you enable a cache, the indexer evaluates changes in your pipeline composition to determine which content can be reused and which needs reprocessing. This section enumerates changes that invalidate the cache outright, followed by changes that trigger incremental processing.

An invalidating change is one where the entire cache is no longer valid. An example of an invalidating change is one where your data source is updated. Here is the complete list of changes to any part of the indexer pipeline that would invalidate your cache:

- Changing the data source type
- Changing data source container
- Changing data source credentials
- Changing data source change detection policy
- Changing data source delete detection policy
- Changing indexer field mappings
- Changing indexer parameters:
  - Parsing Mode
  - Excluded File Name Extensions
  - Indexed File Name Extensions
  - Index storage metadata only for oversized documents
  - Delimited text headers
  - Delimited text delimiter
  - Document Root
  - Image Action (Changes to how images are extracted)

Changes that trigger incremental processing

Incremental processing evaluates your skillset definition and determines which skills to rerun, selectively updating the affected portions of the document tree. Here is the complete list of changes resulting in incremental enrichment:

- Changing the skill type (the OData type of the skill is updated)

- Skill-specific parameters are updated, for example a URL, defaults, or other parameters
- Skill output changes, the skill returns additional or different outputs
- Skill input changes resulting in different ancestry, skill chaining has changed
- Any upstream skill invalidation, if a skill that provides an input to this skill is updated
- Updates to the knowledge store projection location, results in re-projecting documents
- Changes to the knowledge store projections, results in re-projecting documents
- Output field mappings changed on an indexer results in re-projecting documents to the index

APIs used for caching

REST API version `2020-06-30-Preview` or later provides incremental enrichment through additional properties on indexers. Skillsets and data sources can use the generally available version. In addition to the reference documentation, see Configure caching for incremental enrichment for details about order of operations.

- Create or Update Indexer (api-version=2020-06-30-Preview)
- Update Skillset (api-version=2020-06-30) (New URI parameter on the request)
- Reset Skills (api-version=2020-06-30)
- Update Data Source, when called with a preview API version, provides a new parameter named "ignoreResetRequirement", which should be set to true when your update action should not invalidate the cache. Use "ignoreResetRequirement" sparingly as it could lead to unintended inconsistency in your data that will not be detected easily.

Limitations

> **⊗ Caution**
>
> If you're using the **SharePoint Online indexer (Preview)**, you should avoid incremental enrichment. Under certain circumstances, the cache becomes invalid, requiring an **indexer reset and run**, should you choose to reload it.

Next steps

Incremental enrichment is a powerful feature that extends change tracking to skillsets and AI enrichment. Incremental enrichment enables reuse of existing processed content as you iterate over skillset design. As a next step, enable caching on your indexers.

Enable caching for incremental enrichment

Skillset concepts in Azure Cognitive Search

Article • 10/12/2022 • 11 minutes to read

This article is for developers who need a deeper understanding of skillset concepts and composition, and assumes familiarity with the high-level concepts of AI enrichment.

A skillset is a reusable resource in Azure Cognitive Search that's attached to an indexer. It contains one or more skills that call built-in AI or external custom processing over documents retrieved from an external data source.

The following diagram illustrates the basic data flow of skillset execution.

From the onset of skillset processing to its conclusion, skills read from and write to an *enriched document*. Initially, an enriched document is just the raw content extracted from a data source (articulated as the `"/document"` root node). With each skill execution, the enriched document gains structure and substance as skill writes its output as nodes in the graph.

After skillset execution is done, the output of an enriched document finds its way into an index through *output field mappings*. Any raw content that you want transferred intact, from source to an index, is defined through *field mappings*.

To configure enrichment, you'll specify settings in a skillset and indexer.

Skillset definition

A skillset is an array of one or more *skills* that perform an enrichment, such as translating text or OCR on an image file. Skills can be the built-in skills from Microsoft, or custom skills for processing logic that you host externally. A skillset produces enriched documents that are either consumed during indexing or projected to a knowledge store.

Skills have a context, inputs, and outputs:

- Context refers to the scope of the operation, which could be once per document or once for each item in a collection.
- Inputs originate from nodes in an enriched document, where a "source" and "name" identify a given node.
- Output is sent back to the enriched document as a new node. Values are the node "name" and node content. If a node name is duplicated, you can set a target name for disambiguation.

Skill context

Each skill has a context, which can be the entire document (`/document`) or a node lower in the tree (`/document/countries/*`). A context determines:

- The number of times the skill executes, over a single value (once per field, per document), or for context values of type collection, where adding an `/*` results in skill invocation, once for each instance in the collection.
- Output declaration, or where in the enrichment tree the skill outputs are added. Outputs are always added to the tree as children of the context node.
- Shape of the inputs. For multi-level collections, setting the context to the parent collection will affect the shape of the input for the skill. For example if you have an enrichment tree with a list of countries/regions, each enriched with a list of states containing a list of ZIP codes, how you set the context will determine how the input is interpreted.

Context	Input	Shape of Input	Skill Invocation
`/document/countries/*`	`/document/countries/*/states/*/zipcodes/*`	A list of all ZIP codes in the country/region	Once per country/region
`/document/countries/*/states/*`	`/document/countries/*/states/*/zipcodes/*`	A list of ZIP codes in the state	Once per combination of country/region and state

Skill dependencies

Skills can execute independently and in parallel, or sequentially if you feed the output of one skill into another skill. The following example demonstrates two built-in skills that execute in sequence:

- Skill #1 is a Text Split skill that accepts the contents of the "reviews_text" source field as input, and splits that content into "pages" of 5000 characters as output. Splitting large text into smaller chunks can produce better outcomes for skills like sentiment detection.
- Skill #2 is a Sentiment Detection skill accepts "pages" as input, and produces a new field called "Sentiment" as output that contains the results of sentiment analysis.

Notice how the output of the first skill ("pages") is used in sentiment analysis, where "/document/reviews_text/pages/*" is both the context and input. For more information about path formulation, see How to reference annotations.

JSON

```
{
    "skills": [
        {
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
            "name": "#1",
            "description": null,
            "context": "/document/reviews_text",
            "defaultLanguageCode": "en",
            "textSplitMode": "pages",
            "maximumPageLength": 5000,
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/reviews_text"
                }
            ],
            "outputs": [
                {
                    "name": "textItems",
                    "targetName": "pages"
                }
            ]
        },
        {
```

```
            "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
            "name": "#2",
            "description": null,
            "context": "/document/reviews_text/pages/*",
            "defaultLanguageCode": "en",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/reviews_text/pages/*",
                }
            ],
            "outputs": [
                {
                    "name": "sentiment",
                    "targetName": "sentiment"
                },
                {
                    "name": "confidenceScores",
                    "targetName": "confidenceScores"
                },
                {
                    "name": "sentences",
                    "targetName": "sentences"
                }
            ]
        }
      . . .
  ]
}
```

Enrichment tree

An enriched document is a temporary, tree-like data structure created during skillset execution that collects all of the changes introduced through skills. Collectively, enrichments are represented as a hierarchy of addressable nodes. Nodes also include any unenriched fields that are passed in verbatim from the external data source.

An enriched document exists for the duration of skillset execution, but can be cached or sent to a knowledge store.

Initially, an enriched document is simply the content extracted from a data source during *document cracking*, where text and images are extracted from the source and made available for language or image analysis.

The initial content is metadata and the *root node* (`document/content`). The root node is usually a whole document or a normalized image that is extracted from a data source during document cracking. How it's articulated in an enrichment tree varies for each data source type. The following table shows the state of a document entering into the enrichment pipeline for several supported data sources:

Data Source\Parsing Mode	Default	JSON, JSON Lines & CSV
Blob Storage	/document/content /document/normalized_images/* ...	/document/{key1} /document/{key2} ...

Data Source\Parsing Mode	Default	JSON, JSON Lines & CSV
Azure SQL	/document/{column1} /document/{column2} ...	N/A
Azure Cosmos DB	/document/{key1} /document/{key2} ...	N/A

As skills execute, output is added to the enrichment tree as new nodes. If skill execution is over the entire document, nodes are added at the first level under the root.

Nodes can be used as inputs for downstream skills. For example, skills that create content, such as translated strings, could become input for skills that recognize entities or extract key phrases.

Although you can visualize and work with an enrichment tree through the Debug Sessions visual editor, it's mostly an internal structure.

Enrichments are immutable: once created, nodes can't be edited. As your skillsets get more complex, so will your enrichment tree, but not all nodes in the enrichment tree need to make it to the index or the knowledge store.

You can selectively persist just a subset of the enrichment outputs so that you're only keeping what you intend to use. The output field mappings in your indexer definition will determine what content actually gets ingested in the search index. Likewise, if you're creating a knowledge store, you can map outputs into shapes that are assigned to projections.

> ⓘ **Note**
>
> The enrichment tree format enables the enrichment pipeline to attach metadata to even primitive data types. The metadata won't be a valid JSON object, but can be projected into a valid JSON format in projection definitions in a knowledge store. For more information, see **Shaper skill**.

Indexer definition

An indexer has properties and parameters used to configure indexer execution. Among those properties are mappings that set the data path to fields in a search index.

There are two sets of mappings:

- "fieldMappings" map a source field to a search field.
- "outputFieldMappings" map a node in an enriched document to a search field.

The "sourceFieldName" property specifies either a field in your data source or a node in an enrichment tree. The "targetFieldName" property specifies the search field in an index that receives the content.

Enrichment example

Using the hotel reviews skillset as a reference point, this example explains how an enrichment tree evolves through skill execution using conceptual diagrams.

This example also shows:

- How a skill's context and inputs work to determine how many times a skill executes
- What the shape of the input is based on the context

In this example, source fields from a CSV file include customer reviews about hotels ("reviews_text") and ratings ("reviews_rating"). The indexer adds metadata fields from Blob storage, and skills add translated text, sentiment scores, and key phrase detection.

In the hotel reviews example, a "document" within the enrichment process represents a single hotel review.

> **Tip**
>
> You can create a search index and knowledge store for this data in **Azure portal** or through **Postman and the REST APIs**. You can also use **Debug Sessions** for insights into skillset composition, dependencies, and effects on an enrichment tree. Images in this article are pulled from Debug Sessions.

Conceptually, the initial enrichment tree looks as follows:

The root node for all enrichments is `"/document"`. When you're working with blob indexers, the `"/document"` node will have child nodes of `"/document/content"` and `"/document/normalized_images"`. When the data is CSV, as in this example, the column names will map to nodes beneath `"/document"`.

Skill #1: Split skill

When source content consists of large chunks of text, it's helpful to break it into smaller components for greater accuracy of language, sentiment, and key phrase detection. There are two grains available: pages and sentences. A page consists of approximately 5000 characters.

A text split skill is typically first in a skillset.

JSON

```
"@odata.type": "#Microsoft.Skills.Text.SplitSkill",
"name": "#1",
"description": null,
"context": "/document/reviews_text",
"defaultLanguageCode": "en",
"textSplitMode": "pages",
"maximumPageLength": 5000,
"inputs": [
{
    "name": "text",
    "source": "/document/reviews_text"
}
```

```
  ],
  "outputs": [
  {
      "name": "textItems",
      "targetName": "pages"
  }
```

With the skill context of `"/document/reviews_text"`, the split skill executes once for the `reviews_text`. The skill output is a list where the `reviews_text` is chunked into 5000 character segments. The output from the split skill is named `pages` and it's added to the enrichment tree. The `targetName` feature allows you to rename a skill output before being added to the enrichment tree.

The enrichment tree now has a new node placed under the context of the skill. This node is available to any skill, projection, or output field mapping.

To access any of the enrichments added to a node by a skill, the full path for the enrichment is needed. For example, if you want to use the text from the `pages` node as an input to another skill, you'll need to specify it as `"/document/reviews_text/pages/*"`. For more information about paths, see Reference annotations.

Skill #2 Language detection

Hotel review documents include customer feedback expressed in multiple languages. The language detection skill determines which language is used. The result will then be passed to key phrase extraction and sentiment detection (not shown), taking language into consideration when detecting sentiment and phrases.

While the language detection skill is the third (skill #3) skill defined in the skillset, it's the next skill to execute. It doesn't require any inputs so it executes in parallel with the previous skill. Like the split skill that preceded it, the language detection skill is also invoked once for each document. The enrichment tree now has a new node for language.

Skills #3 and #4 (sentiment analysis and key phrase detection)

Customer feedback reflects a range of positive and negative experiences. The sentiment analysis skill analyzes the feedback and assigns a score along a continuum of negative to positive numbers, or neutral if sentiment is undetermined. Parallel to sentiment analysis, key phrase detection identifies and extracts words and short phrases that appear consequential.

Given the context of `/document/reviews_text/pages/*`, both sentiment analysis and key phrase skills will be invoked once for each of the items in the `pages` collection. The output from the skill will be a node under the associated page element.

You should now be able to look at the rest of the skills in the skillset and visualize how the tree of enrichments will continue to grow with the execution of each skill. Some skills, such as the merge skill and the shaper skill, also create new nodes but only use data from existing nodes and don't create net new enrichments.

The colors of the connectors in the tree above indicate that the enrichments were created by different skills and the nodes will need to be addressed individually and won't be part of the object returned when selecting the parent node.

Skill #5 Shaper skill

If output includes a knowledge store, add a Shaper skill as a last step. The Shaper skill creates data shapes out of nodes in an enrichment tree. For example, you might want to consolidate multiple nodes into a single shape. You can then project this shape as a table (nodes become the columns in a table), passing the shape by name to a table projection.

The Shaper skill is easy to work with because it focuses shaping under one skill. Alternatively, you can opt for in-line shaping within individual projections. The Shaper Skill doesn't add or detract from an enrichment tree, so it's not visualized. Instead, you can think of a Shaper skill as the means by which you rearticulate the enrichment tree you already have. Conceptually, this is similar to creating views out of tables in a database.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "name": "#5",
  "description": null,
  "context": "/document",
  "inputs": [
    {
      "name": "name",
      "source": "/document/name"
    },
    {
      "name": "reviews_date",
      "source": "/document/reviews_date"
    },
    {
      "name": "reviews_rating",
      "source": "/document/reviews_rating"
    },
    {
      "name": "reviews_text",
      "source": "/document/reviews_text"
    },
    {
      "name": "reviews_title",
      "source": "/document/reviews_title"
    },
    {
      "name": "AzureSearch_DocumentKey",
      "source": "/document/AzureSearch_DocumentKey"
    },
    {
      "name": "pages",
      "sourceContext": "/document/reviews_text/pages/*",
      "inputs": [
        {
          "name": "Sentiment",
          "source": "/document/reviews_text/pages/*/Sentiment"
```

```
        },
        {
          "name": "LanguageCode",
          "source": "/document/Language"
        },
        {
          "name": "Page",
          "source": "/document/reviews_text/pages/*"
        },
        {
          "name": "keyphrase",
          "sourceContext": "/document/reviews_text/pages/*/Keyphrases/*",
          "inputs": [
            {
              "name": "Keyphrases",
              "source": "/document/reviews_text/pages/*/Keyphrases/*"
            }
          ]
        }
      ]
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "tableprojection"
    }
  ]
}
```

Next steps

With an introduction and example behind you, try creating your first skillset using built-in skills.

Create your first skillset

Debug Sessions in Azure Cognitive Search

Article • 10/20/2022 • 4 minutes to read

Debug Sessions is a visual editor that works with an existing skillset in the Azure portal, exposing the structure and content of a single enriched document, as it's produced by an indexer and skillset for the duration of the session. Because you are working with a live document, the session is interactive - you can identify errors, modify and invoke skill execution, and validate the results in real time. If your changes resolve the problem, you can commit them to a published skillset to apply the fixes globally.

How a debug session works

When you start a session, the search service creates a copy of the skillset, indexer, and a data source containing a single document that will be used to test the skillset. All session state will be saved to a new blob container created by the Azure Cognitive Search service in an Azure Storage account that you provide. The name of the generated container has a prefix of "ms-az-cognitive-search-debugsession". The Azure Storage flow to choose the target storage account where the debug session data is exported always requests a container to be chosen by the user. This is omitted by default to avoid debug session data to be exported by mistake to a customer created container that may have data not related to the debug session.

A cached copy of the enriched document and skillset is loaded into the visual editor so that you can inspect the content and metadata of the enriched document, with the ability to check each document node and edit any aspect of the skillset definition. Any changes made within the session are cached. Those changes will not affect the published skillset unless you commit them. Committing changes will overwrite the production skillset.

If the enrichment pipeline does not have any errors, a debug session can be used to incrementally enrich a document, test and validate each change before committing the changes.

Managing the Debug Session state

The debug session can be run again after its creation and its first run using the **Start** button. It may also be canceled while it is still executing using the **Cancel** button. Finally, it may be deleted using the **Delete** button.

AI Enrichments tab > Skill Graph

The visual editor is organized into tabs and panes. This section introduces the components of the visual editor.

The **Skill Graph** provides a visual hierarchy of the skillset and its order of execution from top to bottom. Skills that are dependent upon the output of other skills are positioned lower in the graph. Skills at the same level in the hierarchy can execute in parallel. Color coded labels of skills in the graph indicate the types of skills that are being executed in the skillset (TEXT or VISION).

Selecting a skill in the graph will display the details of that instance of the skill in the right pane, including its definition, errors or warnings, and execution history. The **Skill Graph** is where you will select which skill to debug or enhance. The details pane to the right is where you edit and explore.

Skill details pane

When you select an object in the **Skill Graph**, the adjacent pane provides interactive work areas in a tabbed layout. An illustration of the details pane can be found in the

previous screenshot.

Skill details include the following areas:

- **Skill Settings** shows a formatted version of the skill definition.
- **Skill JSON Editor** shows the raw JSON document of the definition.
- **Executions** shows the data corresponding to each time a skill was executed.
- **Errors and warnings** shows the messages generated upon session start or refresh.

On Executions or Skill Settings, select the `</>` symbol to open the **Expression Evaluator** used for viewing and editing the expressions of the skills inputs and outputs.

Nested input controls in Skill Settings can be used to build complex shapes for projections, output field mappings for a complex type field, or an input to a skill. When used with the Expression Evaluator, nested inputs provide an easy test and validate expression builder.

Executions pane

A skill can execute multiple times in a skillset for a single document. For example, the OCR skill will execute once for each image extracted from a single document. The Executions pane displays the skill's execution history providing a deeper look into each invocation of the skill.

The execution history enables tracking a specific enrichment back to the skill that generated it. Clicking on a skill input navigates to the skill that generated that input, providing a stack-trace like feature. This allows identification of the root cause of a problem that may manifest in a downstream skill.

When you debug an error with a custom skill, there is the option to generate a request for a skill invocation in the execution history.

AI Enrichments tab > Enriched Data Structure

The **Enriched Data Structure** pane shows the document's enrichments through the skillset, detailing the context for each enrichment and the originating skill. The Expression Evaluator can also be used to view the contents for each enrichment.

Expression Evaluator

Expression Evaluator gives a quick peek into the value of any path. It allows for editing the path and testing the results before updating any of the inputs or context for a skill or projection.

You can open the window from any node or element that shows the `</>` symbol, including parts of a dependency graph or nodes in an enrichment tree.

Expression Evaluator gives you full interactive access for testing skill context, inputs, and checking outputs.

Next steps

Now that you understand the elements of debug sessions, start your first debug session on an existing skillset.

How to debug a skillset

Knowledge store in Azure Cognitive Search

Article • 01/31/2023 • 7 minutes to read

Knowledge store is a data sink created by a Cognitive Search enrichment pipeline that stores AI-enriched content in tables and blob containers in Azure Storage for independent analysis or downstream processing in non-search scenarios like knowledge mining.

If you've used cognitive skills in the past, you already know that enriched content is created by *skillsets*. Skillsets move a document through a sequence of enrichments that invoke atomic transformations, such as recognizing entities or translating text.

Output is always a search index, but it can also be projections in a knowledge store. The two outputs, search index and knowledge store, are mutually exclusive products of the same pipeline. They are derived from the same inputs, but their content is structured, stored, and used in different applications.

Physically, a knowledge store is Azure Storage, either Azure Table Storage, Azure Blob Storage, or both. Any tool or process that can connect to Azure Storage can consume the contents of a knowledge store.

Viewed through Azure portal, a knowledge store looks like any other collection of tables, objects, or files. The following screenshot shows a knowledge store composed of three tables. You can adopt a naming convention, such as a "kstore" prefix, to keep your content together.

Benefits of knowledge store

The primary benefits of a knowledge store are two-fold: flexible access to content, and the ability to shape data.

Unlike a search index that can only be accessed through queries in Cognitive Search, a knowledge store can be accessed by any tool, app, or process that supports connections to Azure Storage. This flexibility opens up new scenarios for consuming the analyzed and enriched content produced by an enrichment pipeline.

The same skillset that enriches data can also be used to shape data. Some tools like Power BI work better with tables, whereas a data science workload might require a complex data structure in a blob format. Adding a Shaper skill to a skillset gives you control over the shape of your data. You can then pass these shapes to projections, either tables or blobs, to create physical data structures that align with the data's intended use.

The following video explains both of these benefits and more.
https://www.youtube-nocookie.com/embed/XWzLBP8iWqg?version=3

Knowledge store definition

A knowledge store is defined inside a skillset definition and it has two components:

- A connection string to Azure Storage

- **Projections** that determine whether the knowledge store consists of tables, objects or files. The projections element is an array. You can create multiple sets of table-object-file combinations within one knowledge store.

```JSON
"knowledgeStore": {
    "storageConnectionString":"<YOUR-AZURE-STORAGE-ACCOUNT-CONNECTION-STRING>",
    "projections":[
       {
          "tables":[ ],
          "objects":[ ],
          "files":[ ]
       }
    ]
}
```

The type of projection you specify in this structure determines the type of storage used by knowledge store, but not its structure. Fields in tables, objects, and files are determined by Shaper skill output if you're creating the knowledge store programmatically, or by the Import data wizard if you're using the portal.

- `tables` project enriched content into Table Storage. Define a table projection when you need tabular reporting structures for inputs to analytical tools or export as data frames to other data stores. You can specify multiple `tables` within the same projection group to get a subset or cross section of enriched documents. Within the same projection group, table relationships are preserved so that you can work with all of them.

  Projected content is not aggregated or normalized. The following screenshot shows a table, sorted by key phrase, with the parent document indicated in the adjacent column. In contrast with data ingestion during indexing, there is no linguistic analysis or aggregation of content. Plural forms and differences in casing are considered unique instances.

Content.metadata_storage_name	Content.KeyPhrases
Cognitive Services and Content Intelligence.pptx	Computer Vision
10-K-FY16.html	computing device
10-K-FY16.html	computing devices
MSFT_FY17_10K.docx	computing devices
10-K-FY16.html	Computing segment
Cognitive Services and Bots (spanish).pdf	confianza

- `objects` project JSON document into Blob storage. The physical representation of an `object` is a hierarchical JSON structure that represents an enriched document.
- `files` project image files into Blob storage. A `file` is an image extracted from a document, transferred intact to Blob storage. Although it is named "files", it shows up in Blob Storage, not file storage.

Create a knowledge store

To create knowledge store, use the portal or an API.

You'll need Azure Storage, a skillset, and an indexer. Because indexers require a search index, you'll also need to provide an index definition.

Go with the portal approach for the fastest route to a finished knowledge store. Or, choose the REST API for a deeper understanding of how objects are defined and related.

Azure portal

Create your first knowledge store in four steps using the **Import data** wizard.

1. Define a data source that contains the data you want to enrich.
2. Define a skillset. The skillset specifies enrichment steps and the knowledge store.
3. Define an index schema. You might not need one, but indexers require it. The wizard can infer an index.
4. Complete the wizard. Data extraction, enrichment, and knowledge store creation occur in this last step.

The wizard automates several tasks. Specifically, both shaping and projections (definitions of physical data structures in Azure Storage) are created for you.

Connect with apps

Once enriched content exists in storage, any tool or technology that connects to Azure Storage can be used to explore, analyze, or consume the contents. The following list is a start:

- Storage Explorer or Storage browser (preview) in Azure portal to view enriched document structure and content. Consider this as your baseline tool for viewing knowledge store contents.
- Power BI for reporting and analysis.
- Azure Data Factory for further manipulation.

Content lifecycle

Each time you run the indexer and skillset, the knowledge store is updated if the skillset or underlying source data has changed. Any changes picked up by the indexer are propagated through the enrichment process to the projections in the knowledge store, ensuring that your projected data is a current representation of content in the originating data source.

> **Note**
>
> While you can edit the data in the projections, any edits will be overwritten on the next pipeline invocation, assuming the document in source data is updated.

Changes in source data

For data sources that support change tracking, an indexer will process new and changed documents, and bypass existing documents that have already been processed. Timestamp information varies by data source, but in a blob container, the indexer looks at the `lastmodified` date to determine which blobs need to be ingested.

Changes to a skillset

If you are making changes to a skillset, you should enable caching of enriched documents to reuse existing enrichments where possible.

Without incremental caching, the indexer will always process documents in order of the high water mark, without going backwards. For blobs, the indexer would process blobs sorted by `lastModified`, regardless of any changes to indexer settings or the skillset. If you change a skillset, previously processed documents aren't updated to reflect the new skillset. Documents processed after the skillset change will use the new skillset, resulting in index documents being a mix of old and new skillsets.

With incremental caching, and after a skillset update, the indexer will reuse any enrichments that are unaffected by the skillset change. Upstream enrichments are pulled from cache, as are any enrichments that are independent and isolated from the skill that was changed.

Deletions

Although an indexer creates and updates structures and content in Azure Storage, it does not delete them. Projections continue to exist even when the indexer or skillset is deleted. As the owner of the storage account, you should delete a projection if it is no longer needed.

Next steps

Knowledge store offers persistence of enriched documents, useful when designing a skillset, or the creation of new structures and content for consumption by any client applications capable of accessing an Azure Storage account.

The simplest approach for creating enriched documents is through the portal, but you can also use Postman and the REST API, which is more useful if you want insight into how objects are created and referenced programmatically.

Create a knowledge store using Postman and REST

Additional resources

Documentation

Shaper cognitive skill - Azure Cognitive Search

Extract metadata and structured information from unstructured data and shape it as a complex type in an AI enrichment pipeline in Azure Cognitive Search.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Create a knowledge store using REST - Azure Cognitive Search

Use the REST API and Postman to create an Azure Cognitive Search knowledge store for persisting AI enrichments from skillset.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom

Web API skill to integrate your custom code.

Projection concepts - Azure Cognitive Search

Introduces projection concepts and best practices. If you are creating a knowledge store in Cognitive Search, projections will determine the type, quantity, and composition of objects in Azure Storage.

Define projections - Azure Cognitive Search

Learn how to define table, object, and file projections in a knowledge store by reviewing syntax and examples.

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Key Phrase Extraction cognitive skill - Azure Cognitive Search

Evaluates unstructured text, and for each record, returns a list of key phrases in an AI enrichment pipeline in Azure Cognitive Search.

Show 5 more

Knowledge store "projections" in Azure Cognitive Search

Article • 11/15/2022 • 6 minutes to read

Projections are the physical tables, objects, and files in a **knowledge store** that accept content from a Cognitive Search AI enrichment pipeline. If you're creating a knowledge store, defining and shaping projections is most of the work.

This article introduces projection concepts and workflow so that you have some background before you start coding.

Projections are defined in Cognitive Search skillsets, but the end results are the table, object, and image file projections in Azure Storage.

Types of projections and usage

A knowledge store is a logical construction that's physically expressed as a loose collection of tables, JSON objects, or binary image files in Azure Storage.

Projection	Storage	Usage

Projection	Storage	Usage
Tables	Azure Table Storage	Used for data that's best represented as rows and columns, or whenever you need granular representations of your data (for example, as data frames). Table projections allow you to define a schematized shape, using a Shaper skill or use inline shaping to specify columns and rows. You can organize content into multiple tables based on familiar normalization principles. Tables that are in the same group are automatically related.
Objects	Azure Blob Storage	Used when you need the full JSON representation of your data and enrichments in one JSON document. As with table projections, only valid JSON objects can be projected as objects, and shaping can help you do that.
Files	Azure Blob Storage	Used when you need to save normalized, binary image files.

Projection definition

Projections are specified under the "knowledgeStore" property of a skillset. Projection definitions are used during indexer invocation to create and load objects in Azure Storage with enriched content. If you are unfamiliar with these concepts, start with AI enrichment for an introduction.

The following example illustrates the placement of projections under knowledgeStore, and the basic construction. The name, type, and content source make up a projection definition.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
      {
        "tables": [
          { "tableName": "ks-museums-main", "generatedKeyName": "ID",
"source": "/document/tableprojection" },
          { "tableName": "ks-museumEntities", "generatedKeyName":
"ID","source": "/document/tableprojection/Entities/*" }
        ],
        "objects": [
          { "storageContainer": "ks-museums", "generatedKeyName": "ID",
"source": "/document/objectprojection" }
        ],
        "files": [ ]
```

```
        }
    ]
```

Projection groups

Projections are an array of complex collections, which means that you can specify multiple sets of each type. It's common to use just one projection group, but you might use multiple if storage requirements include supporting different tools and scenarios. For example, you might use one group for design and debug of a skillset, while a second set collects output used for an online app, with a third for data science workloads.

The same skillset output is used to populate all groups under projections. The following example shows two.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [],
            "objects": [],
            "files": []
        },
        {
            "tables": [],
            "objects": [],
            "files": []
        }
    ]
}
```

Projection groups have the following key characteristics of mutual exclusivity and relatedness.

Principle	Description
Mutual exclusivity	Each group is fully isolated from other groups to support different data shaping scenarios. For example, if you are testing different table structures and combinations, you would put each set in a different projection group for AB testing. Each group obtains data from the same source (enrichment tree) but is fully isolated from the table-object-file combination of any peer projection groups.

Principle	Description
Relatedness	Within a projection group, content in tables, objects, and files are related. Knowledge store uses generated keys as reference points to a common parent node. For example, consider a scenario where you have a document containing images and text. You could project the text to tables and the images to binary files, and both tables and objects will have a column/property containing the file URL.

Projection "source"

The source parameter is the third component of a projection definition. Because projections store data from an AI enrichment pipeline, the source of a projection is always the output of a skill. As such, output might be a single field (for example, a field of translated text), but often it's a reference to a data shape.

Data shapes come from your skillset. Among all of the built-in skills provided in Cognitive Search, there is a utility skill called the **Shaper skill** that's used to create data shapes. You can include Shaper skills (as many as you need) to support the projections in the knowledge store.

Shapes are frequently used with table projections, where the shape not only specifies which rows go into the table, but also which columns are created (you can also pass a shape to an object projection).

Shapes can be complex and it's out of scope to discuss them in depth here, but the following example briefly illustrates a basic shape. The output of the Shaper skill is specified as the source of a table projection. Within the table projection itself will be columns for "metadata-storage_path", "reviews_text", "reviews_title", and so forth, as specified in the shape.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "ShaperForTables",
    "description": null,
    "context": "/document",
    "inputs": [
        {
            "name": "metadata_storage_path",
            "source": "/document/metadata_storage_path",
            "sourceContext": null,
            "inputs": []
        },
        {
          "name": "reviews_text",
```

```
            "source": "/document/reviews_text"
          },
          {
            "name": "reviews_title",
            "source": "/document/reviews_title"
          },
          {
            "name": "reviews_username",
            "source": "/document/reviews_username"
          },
      ],
      "outputs": [
        {
          "name": "output",
          "targetName": "mytableprojection"
        }
      ]
  }
```

Projection lifecycle

Projections have a lifecycle that is tied to the source data in your data source. As source data is updated and reindexed, projections are updated with the results of the enrichments, ensuring your projections are eventually consistent with the data in your data source. However, projections are also independently stored in Azure Storage. They will not be deleted when the indexer or the search service itself is deleted.

Consume in apps

After the indexer is run, connect to projections and consume the data in other apps and workloads.

- Use Azure portal to verify object creation and content in Azure Storage.
- Use Power BI for data exploration. This tool works best when the data is in Azure Table Storage. Within Power BI, you can manipulate data into new tables that are easier to query and analyze.
- Use enriched data in blob container in a data science pipeline. For example, you can load the data from blobs into a Pandas DataFrame.
- Finally, if you need to export your data from the knowledge store, Azure Data Factory has connectors to export the data and land it in the database of your choice.

Checklist for getting started

Recall that projections are exclusive to knowledge stores, and are not used to structure a search index.

1. In Azure Storage, get a connection string from **Access Keys** and verify the account is StorageV2 (general purpose V2).

2. While in Azure Storage, familiarize yourself with existing content in containers and tables so that you choose non-conflicting names for the projections. A knowledge store is a loose collection of tables and containers. Consider adopting a naming convention to keep track of related objects.

3. In Cognitive Search, enable enrichment caching (preview) in the indexer and then run the indexer to execute the skillset and populate the cache. This is a preview feature, so be sure to use the preview REST API (api-version=2020-06-30-preview or later) on the indexer request. Once the cache is populated, you can modify projection definitions in a knowledge store free of charge (as long as the skills themselves are not modified).

4. In your code, all projections are defined solely in a skillset. There are no indexer properties (such as field mappings or output field mappings) that apply to projections. Within a skillset definition, you will focus on two areas: knowledgeStore property and skills array.

   a. Under knowledgeStore, specify table, object, file projections in the `projections` section. Object type, object name, and quantity (per the number of projections you define) are determined in this section.

   b. From the skills array, determine which skill outputs will be referenced in the `source` of each projection. All projections have a source. The source can be the output of an upstream skill, but is often the output of a Shaper skill. The composition of your projection is determined through shapes.

5. If you're adding projections to an existing skillset, update the skillset and run the indexer.

6. Check your results in Azure Storage. On subsequent runs, avoid naming collisions by deleting objects in Azure Storage or changing project names in the skillset.

7. If you are using Table projections check Understanding the Table Service data model and Scalability and performance targets for Table storage to make sure your data requirements are within Table storage documented limits.

Next steps

Review syntax and examples for each projection type.

Define projections in a knowledge store

Security overview for Azure Cognitive Search

Article • 05/15/2023

This article describes the security features in Azure Cognitive Search that protect data and operations.

Data flow (network traffic patterns)

A Cognitive Search service is hosted on Azure and is typically accessed by client applications over public network connections. While that pattern is predominant, it's not the only traffic pattern that you need to care about. Understanding all points of entry as well as outbound traffic is necessary background for securing your development and production environments.

Cognitive Search has three basic network traffic patterns:

- Inbound requests made by a client to the search service (the predominant pattern)
- Outbound requests issued by the search service to other services on Azure and elsewhere
- Internal service-to-service requests over the secure Microsoft backbone network

Inbound traffic

Inbound requests that target a search service endpoint can be characterized as:

- Create or manage indexes, indexers, data sources, skillsets, and synonym maps
- Invoke indexer or skillset execution
- Load or query an index

You can review the REST APIs to understand the full range of inbound requests that are handled by a search service.

At a minimum, all inbound requests must be authenticated:

- Key-based authentication is the default. Inbound requests that include a valid API key are accepted by the search service as originating from a trusted party.
- Alternatively, you can use Azure Active Directory and role-based access control for data plane operations.

Additionally, you can add network security features to further restrict access to the endpoint. You can create either inbound rules in an IP firewall, or create private endpoints that fully shield your search service from the public internet.

Outbound traffic

Outbound requests from a search service to other applications are typically made by indexers for text-based indexing and some aspects of AI enrichment. Outbound requests include both read and write operations.

The following list is a full enumeration of the outbound requests that can be made by a search service. A search service makes requests on its own behalf, and on the behalf of an indexer or custom skill:

- Indexers read from external data sources.
- Indexers write to Azure Storage when creating knowledge stores, persisting cached enrichments, and persisting debug sessions.
- If you're using custom skills, custom skills connect to an external Azure function or app to run external code that's hosted off-service. The request for external processing is sent during skillset execution.
- If you're using customer-managed keys, the service connects to an external Azure Key Vault for a customer-managed key used to encrypt and decrypt sensitive data.

Outbound connections can be made using a resource's full access connection string that includes a key or a database login, or an Azure AD login (a managed identity) if you're using Azure Active Directory.

If your Azure resource is behind a firewall, you'll need to create rules that admit search service requests. For resources protected by Azure Private Link, you can create a shared private link that an indexer uses to make its connection.

Exception for same-region search and storage services

If Storage and Search are in the same region, network traffic is routed through a private IP address and occurs over the Microsoft backbone network. Because private IP addresses are used, you can't configure IP firewalls or a private endpoint for network security. Instead, use the trusted service exception as an alternative when both services are in the same region.

Internal traffic

Internal requests are secured and managed by Microsoft. You can't configure or control these connections. If you're locking down network access, no action on your part is required because internal traffic isn't customer-configurable.

Internal traffic consists of:

- Service-to-service calls for tasks like authentication and authorization through Azure Active Directory, resource logging sent to Azure Monitor, and private endpoint connections that utilize Azure Private Link.
- Requests made to Cognitive Services APIs for built-in skills.
- Requests made to the machine learning models that support semantic search.

Network security

Network security protects resources from unauthorized access or attack by applying controls to network traffic. Azure Cognitive Search supports networking features that can be your first line of defense against unauthorized access.

Inbound connection through IP firewalls

A search service is provisioned with a public endpoint that allows access using a public IP address. To restrict which traffic comes through the public endpoint, create an inbound firewall rule that admits requests from a specific IP address or a range of IP addresses. All client connections must be made through an allowed IP address, or the connection is denied.

You can use the portal to configure firewall access.

Alternatively, you can use the management REST APIs. Starting with API version 2020-03-13, with the IpRule parameter, you can restrict access to your service by identifying IP addresses, individually or in a range, that you want to grant access to your search service.

Inbound connection to a private endpoint (network isolation, no Internet traffic)

For more stringent security, you can establish a private endpoint for Azure Cognitive Search allows a client on a virtual network to securely access data in a search index over a Private Link.

The private endpoint uses an IP address from the virtual network address space for connections to your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. A VNET allows for secure communication among resources, with your on-premises network as well as the Internet.

While this solution is the most secure, using more services is an added cost so be sure you have a clear understanding of the benefits before diving in. For more information about costs, see the pricing page. For more information about how these components work together, watch this video. Coverage of the private endpoint option starts at 5:48 into the video. For instructions on how to set up the endpoint, see Create a Private Endpoint for Azure Cognitive Search.

Authentication

Once a request is admitted to the search service, it must still undergo authentication and authorization that determines whether the request is permitted. Cognitive Search supports two approaches:

- Azure AD authentication establishes the caller (and not the request) as the authenticated identity. An Azure role assignment determines the allowed operation.
- Key-based authentication is performed on the request (not the calling app or user) through an API key, where the key is a string composed of randomly generated numbers and letters that prove the request is from a trustworthy source. Keys are

required on every request. Submission of a valid key is considered proof the request originates from a trusted entity.

You can use both authentication methods, or disable an approach that you don't want to use.

Authorization

Cognitive Search provides authorization models for service management and content management.

Authorize service management

Resource management is authorized through Azure role-based access control (Azure RBAC). Azure RBAC is the authorization system for Azure Resource Manager.

In Azure Cognitive Search, Resource Manager is used to create or delete the service, manage API keys, scale the service, and configure security. As such, Azure role assignments will determine who can perform those tasks, regardless of whether they're using the portal, PowerShell, or the Management REST APIs.

Three basic roles (Owner, Contributor, Reader) apply to search service administration. Role assignments can be made using any supported methodology (portal, PowerShell, and so forth) and are honored service-wide.

> **Note**
>
> Using Azure-wide mechanisms, you can lock a subscription or resource to prevent accidental or unauthorized deletion of your search service by users with admin rights. For more information, see **Lock resources to prevent unexpected deletion**.

Authorize access to content

Content management refers to the objects created and hosted on a search service.

- For Azure AD authorization, use Azure role assignments to establish read-write access to your search service.
- For key-based authorization, an API key and a qualified endpoint determine access. An endpoint might be the service itself, the indexes collection, a specific index, a documents collection, or a specific document. When chained together, the

endpoint, the operation (for example, a create or update request) and the type of key (admin or query) authorize access to content and operations.

Restricting access to indexes

Using Azure roles, you can set permissions on individual indexes as long as it's done programmatically.

Using keys, anyone with an admin key to your service can read, modify, or delete any index in the same service. For protection against accidental or malicious deletion of indexes, your in-house source control for code assets is the solution for reversing an unwanted index deletion or modification. Azure Cognitive Search has failover within the cluster to ensure availability, but it doesn't store or execute your proprietary code used to create or load indexes.

For multitenancy solutions requiring security boundaries at the index level, it's common to handle index isolation in the middle tier in your application code. For more information about the multitenant use case, see Design patterns for multitenant SaaS applications and Azure Cognitive Search.

Restricting access to documents

User permissions at the document level, also known as "row-level security", isn't natively supported in Cognitive Search. If you import data from an external system that provides row-level security, such as Azure Cosmos DB, those permissions won't transfer with the data as its being indexed by Cognitive Search.

If you require permissioned access over content in search results, there's a technique for applying filters that include or exclude documents based on user identity. This workaround adds a string field in the data source that represents a group or user identity, which you can make filterable in your index. The following table describes two approaches for trimming search results of unauthorized content.

Approach	Description
Security trimming based on identity filters	Documents the basic workflow for implementing user identity access control. It covers adding security identifiers to an index, and then explains filtering against that field to trim results of prohibited content.
Security trimming based on Azure Active Directory identities	This article expands on the previous article, providing steps for retrieving identities from Azure Active Directory (Azure AD), one of the free services in the Azure cloud platform.

Data residency

When you set up a search service, you choose a location or region that determines where customer data is stored and processed. Azure Cognitive Search won't store customer data outside of your specified region unless you configure a feature that has a dependency on another Azure resource, and that resource is provisioned in a different region.

Currently, the only external resource that a search service writes customer data to is Azure Storage. The storage account is one that you provide, and it could be in any region. A search service will write to Azure Storage if you use any of the following features: enrichment cache, debug session, knowledge store.

Exceptions to data residency commitments

Object names will be stored and processed outside of your selected region or location. Customers shouldn't place any sensitive data in name fields or create applications designed to store sensitive data in these fields. This data appears in the telemetry logs used by Microsoft to provide support for the service. Object names include names of indexes, indexers, data sources, skillsets, resources, containers, and key vault store.

Telemetry logs are retained for one and a half years. During that period, Microsoft might access and reference object names under the following conditions:

- Diagnose an issue, improve a feature, or fix a bug. In this scenario, data access is internal only, with no third-party access.
- During support, this information may be used to provide quick resolution to issues and escalate product team if needed

Data protection

At the storage layer, data encryption is built in for all service-managed content saved to disk, including indexes, synonym maps, and the definitions of indexers, data sources, and skillsets. Service-managed encryption applies to both long-term data storage and temporary data storage.

Optionally, you can add customer-managed keys (CMK) for supplemental encryption of indexed content for double encryption of data at rest. For services created after August 1 2020, CMK encryption extends to short-term data on temporary disks.

Data in transit

In Azure Cognitive Search, encryption starts with connections and transmissions. For search services on the public internet, Azure Cognitive Search listens on HTTPS port 443. All client-to-service connections use TLS 1.2 encryption. Earlier versions (1.0 or 1.1) aren't supported.

Data at rest

For data handled internally by the search service, the following table describes the data encryption models. Some features, such as knowledge store, incremental enrichment, and indexer-based indexing, read from or write to data structures in other Azure Services. Services that have a dependency on Azure Storage can use the encryption features of that technology.

Model	Keys	Requirements	Restrictions	Applies to
server-side encryption	Microsoft-managed keys	None (built-in)	None, available on all tiers, in all regions, for content created after January 24, 2018.	Content (indexes and synonym maps) and definitions (indexers, data sources, skillsets), on data disks and temporary disks
server-side encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in specific regions, for content created after August 1, 2020.	Content (indexes and synonym maps) on data disks
server-side full encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in all regions, on search services after May 13, 2021.	Content (indexes and synonym maps) on data disks and temporary disks

Service-managed keys

Service-managed encryption is a Microsoft-internal operation that uses 256-bit AES encryption. It occurs automatically on all indexing, including on incremental updates to indexes that aren't fully encrypted (created before January 2018).

Service-managed encryption applies to all content on long-term and short-term storage.

Customer-managed keys (CMK)

Customer-managed keys require another billable service, Azure Key Vault, which can be in a different region, but under the same subscription, as Azure Cognitive Search.

CMK support was rolled out in two phases. If you created your search service during the first phase, CMK encryption was restricted to long-term storage and specific regions. Services created in the second phase, after May 2021, can use CMK encryption in any region. As part of the second wave rollout, content is CMK-encrypted on both long-term and short-term storage. For more information about CMK support, see full double encryption.

Enabling CMK encryption will increase index size and degrade query performance. Based on observations to date, you can expect to see an increase of 30-60 percent in query times, although actual performance will vary depending on the index definition and types of queries. Because of the negative performance impact, we recommend that you only enable this feature on indexes that really require it. For more information, see Configure customer-managed encryption keys in Azure Cognitive Search.

Security administration

Manage API keys

Reliance on API key-based authentication means that you should have a plan for regenerating the admin key at regular intervals, per Azure security best practices. There are a maximum of two admin keys per search service. For more information about securing and managing API keys, see Create and manage api-keys.

Activity and resource logs

Cognitive Search doesn't log user identities so you can't refer to logs for information about a specific user. However, the service does log create-read-update-delete operations, which you might be able to correlate with other logs to understand the agency of specific actions.

Using alerts and the logging infrastructure in Azure, you can pick up on query volume spikes or other actions that deviate from expected workloads. For more information about setting up logs, see Collect and analyze log data and Monitor query requests.

Certifications and compliance

Azure Cognitive Search participates in regular audits, and has been certified against many global, regional, and industry-specific standards for both the public cloud and Azure Government. For the complete list, download the **Microsoft Azure Compliance Offerings** whitepaper from the official Audit reports page.

For compliance, you can use Azure Policy to implement the high-security best practices of Microsoft cloud security benchmark. The Microsoft cloud security benchmark is a collection of security recommendations, codified into security controls that map to key actions you should take to mitigate threats to services and data. There are currently 12 security controls, including Network Security, Logging and Monitoring, and Data Protection.

Azure Policy is a capability built into Azure that helps you manage compliance for multiple standards, including those of Microsoft cloud security benchmark. For well-known benchmarks, Azure Policy provides built-in definitions that provide both criteria and an actionable response that addresses noncompliance.

For Azure Cognitive Search, there's currently one built-in definition. It's for resource logging. You can assign a policy that identifies search services that are missing resource logging, and then turn it on. For more information, see Azure Policy Regulatory Compliance controls for Azure Cognitive Search.

Watch this video

Watch this fast-paced video for an overview of the security architecture and each feature category.
https://learn.microsoft.com/Shows/AI-Show/Azure-Cognitive-Search-Whats-new-in-security/player

See also

- Azure security fundamentals
- Azure Security
- Microsoft Defender for Cloud

Indexer access to content protected by Azure network security

Article • 05/14/2023

If your search application requirements include an Azure virtual network, this concept article explains how a search indexer can access content that's protected by network security. It describes the outbound traffic patterns and indexer execution environments. It also covers the network protections supported by Cognitive Search and factors that might influence your security strategy. Finally, because Azure Storage is used for both data access and persistent storage, this article also covers network considerations that are specific to search and storage connectivity.

Looking for step-by-step instructions instead? See How to configure firewall rules to allow indexer access or How to make outbound connections through a private endpoint.

Resources accessed by indexers

Azure Cognitive Search indexers can make outbound calls to various Azure resources during execution. An indexer makes outbound calls in three situations:

- Connecting to external data sources during indexing
- Connecting to external, encapsulated code through a skillset that includes custom skills
- Connecting to Azure Storage during skillset execution to cache enrichments, save debug session state, or write to a knowledge store

A list of all possible Azure resource types that an indexer might access in a typical run are listed in the table below.

Resource	Purpose within indexer run
Azure Storage (blobs, ADLS Gen 2, files, tables)	Data source
Azure Storage (blobs, tables)	Skillsets (caching enrichments, debug sessions, knowledge store projections)
Azure Cosmos DB (various APIs)	Data source
Azure SQL Database	Data source
SQL Server on Azure virtual machines	Data source

Resource	Purpose within indexer run
SQL Managed Instance	Data source
Azure Functions	Attached to a skillset and used to host for custom web API skills

> **Note**
>
> An indexer also connects to Cognitive Services for built-in skills. However, that connection is made over the internal network and isn't subject to any network provisions under your control.

Supported network protections

Your Azure resources could be protected using any number of the network isolation mechanisms offered by Azure. Depending on the resource and region, Cognitive Search indexers can make outbound connections through IP firewalls and private endpoints, subject to the limitations indicated in the following table.

Resource	IP restriction	Private endpoint
Azure Storage for text-based indexing (blobs, ADLS Gen 2, files, tables)	Supported only if the storage account and search service are in different regions.	Supported
Azure Storage for AI enrichment (caching, debug sessions, knowledge store)	Supported only if the storage account and search service are in different regions.	Supported
Azure Cosmos DB for NoSQL	Supported	Supported
Azure Cosmos DB for MongoDB	Supported	Unsupported
Azure Cosmos DB for Apache Gremlin	Supported	Unsupported
Azure SQL Database	Supported	Supported
SQL Server on Azure virtual machines	Supported	N/A
SQL Managed Instance	Supported	N/A

Resource	IP restriction	Private endpoint
Azure Functions	Supported	Supported, only for certain tiers of Azure functions

Indexer execution environment

Azure Cognitive Search has the concept of an *indexer execution environment* that optimizes processing based on the characteristics of the job. There are two environments. If you're using an IP firewall to control access to Azure resources, knowing about execution environments will help you set up an IP range that is inclusive of both.

For any given indexer run, Azure Cognitive Search determines the best environment in which to run the indexer. Depending on the number and types of tasks assigned, the indexer will run in one of two environments:

- A *private execution environment* that's internal to a search service.

  Indexers running in the private environment share computing resources with other indexing and query workloads on the same search service. Typically, only indexers that perform text-based indexing (without skillsets) run in this environment.

- A *multi-tenant environment* that's managed and secured by Microsoft at no extra cost. It isn't subject to any network provisions under your control.

  This environment is used to offload computationally intensive processing, leaving service-specific resources available for routine operations. Examples of resource-intensive indexer jobs include attaching skillsets, processing large documents, or processing a high volume of documents.

The following section explains the IP configuration for admitting requests from either execution environment.

Setting up IP ranges for indexer execution

If the Azure resource that provides source data exists behind a firewall, you'll need inbound rules that admit indexer connections for all of the IPs from which an indexer request can originate. The IPs include the one used by the search service and the multi-tenant environment.

- To obtain the IP address of the search service (and the private execution environment), you'll use `nslookup` (or `ping`) the fully qualified domain name (FQDN) of your search service. The FQDN of a search service in the public cloud would be `<service-name>.search.windows.net`.

- To obtain the IP addresses of the multi-tenant environments within which an indexer might run, you'll use the `AzureCognitiveSearch` service tag.

  Azure service tags have a published range of IP addresses for each service. You can find these IPs using the discovery API or a downloadable JSON file. IP ranges are allocated by region, so check your search service region before you start.

When setting the IP rule for the multi-tenant environment, certain SQL data sources support a simple approach for IP address specification. Instead of enumerating all of the IP addresses in the rule, you can create a Network Security Group rule that specifies the `AzureCognitiveSearch` service tag.

You can specify the service tag if your data source is either:

- SQL Server on Azure virtual machines

- SQL Managed Instances

Notice that if you specified the service tag for the multi-tenant environment IP rule, you'll still need an explicit inbound rule for the private execution environment (meaning the search service itself), as obtained through `nslookup`.

Choosing a connectivity approach

When integrating Azure Cognitive Search into a solution that runs on a virtual network, consider the following constraints:

- An indexer can't make a direct connection to a virtual network service endpoint. Public endpoints with credentials, private endpoints, trusted service, and IP addressing are the only supported methodologies for indexer connections.

- A search service always runs in the cloud and can't be provisioned into a specific virtual network, running natively on a virtual machine. This functionality won't be offered by Azure Cognitive Search.

Given the above constrains, your choices for achieving search integration in a virtual network are:

- Configure an inbound firewall rule on your Azure PaaS resource that admits indexer requests for data.
- Configure an outbound connection from Search that makes indexer connections using a private endpoint.

  For a private endpoint, the search service connection to your protected resource is through a *shared private link*. A shared private link is an Azure Private Link resource that's created, managed, and used from within Cognitive Search. If your resources are fully locked down (running on a protected virtual network, or otherwise not available over a public connection), a private endpoint is your only choice.

  Connections through a private endpoint must originate from the search service's private execution environment. To meet this requirement, you'll have to disable multi-tenant execution. This step is described in Make outbound connections through a private endpoint.

Configuring an IP firewall is free. A private endpoint, which is based on Azure Private Link, has a billing impact.

Working with a private endpoint

This section summarizes the main steps for setting up a private endpoint for outbound indexer connections. This summary might help you decide whether a private endpoint is the best choice for your scenario. Detailed steps are covered in How to make outbound connections through a private endpoint.

Billing impact of Azure Private Link

- A shared private link requires a billable search service, where the minimum tier is either Basic for text-based indexing or Standard 2 (S2) for skills-based indexing. See tier limits on the number of private endpoints for details.
- Inbound and outbound connections are subject to Azure Private Link pricing.

Step 1: Create a private endpoint to the secure resource

You'll create a shared private link using either the portal pages of your search service or through the Management API.

In Azure Cognitive Search, your search service must be at least the Basic tier for text-based indexers, and S2 for indexers with skillsets.

A private endpoint connection will accept requests from the private indexer execution environment, but not the multi-tenant environment. You'll need to disable multi-tenant execution as described in step 3 to meet this requirement.

Step 2: Approve the private endpoint connection

When the (asynchronous) operation that creates a shared private link resource completes, a private endpoint connection will be created in a "Pending" state. No traffic flows over the connection yet.

You'll need to locate and approve this request on your secure resource. Depending on the resource, you can complete this task using Azure portal. Otherwise, use the Private Link Service REST API.

Step 3: Force indexers to run in the "private" environment

For private endpoint connections, it's mandatory to set the `executionEnvironment` of the indexer to `"Private"`. This step ensures that all indexer execution is confined to the private environment provisioned within the search service.

This setting is scoped to an indexer and not the search service. If you want all indexers to connect over private endpoints, each one must have the following configuration:

JSON

```
{
  "name" : "myindexer",
  ... other indexer properties
  "parameters" : {
      ... other parameters
      "configuration" : {
        ... other configuration properties
        "executionEnvironment": "Private"
      }
    }
}
```

Once you have an approved private endpoint to a resource, indexers that are set to be *private* attempt to obtain access via the private link that was created and approved for the Azure resource.

Azure Cognitive Search will validate that callers of the private endpoint have appropriate Azure RBAC role permissions. For example, if you request a private endpoint connection to a storage account with read-only permissions, this call will be rejected.

If the private endpoint isn't approved, or if the indexer didn't use the private endpoint connection, you'll find a `transientFailure` error message in indexer execution history.

Access to a network-protected storage account

A search service stores indexes and synonym lists. For other features that require storage, Cognitive Search takes a dependency on Azure Storage. Enrichment caching, debug sessions, and knowledge stores fall into this category. The location of each service, and any network protections in place for storage, will determine your data access strategy.

Same-region services

In Azure Storage, access through a firewall requires that the request originates from a different region. If Azure Storage and Azure Cognitive Search are in the same region, you can bypass the IP restrictions on the storage account by accessing data under the system identity of the search service.

There are two options for supporting data access using the system identity:

- Configure search to run as a trusted service and use the trusted service exception in Azure Storage.
- Configure a resource instance rule in Azure Storage that admits inbound requests from an Azure resource.

The above options depend on Azure Active Directory for authentication, which means that the connection must be made with an Azure AD login. Currently, only a Cognitive Search system-assigned managed identity is supported for same-region connections through a firewall.

Services in different regions

When search and storage are in different regions, you can use the previously mentioned options or set up IP rules that admit requests from your service. Depending on the workload, you might need to set up rules for multiple execution environments as described in the next section.

Next steps

Now that you're familiar with indexer data access options for solutions deployed in an Azure virtual network, review either of the following how-to articles as your next step:

- How to make indexer connections to a private endpoint
- How to make indexer connections through an IP firewall

Azure Policy Regulatory Compliance controls for Azure Cognitive Search

Article • 03/08/2023 • 10 minutes to read

If you are using Azure Policy to enforce the recommendations in Microsoft cloud security benchmark, then you probably already know that you can create policies for identifying and fixing non-compliant services. These policies might be custom, or they might be based on built-in definitions that provide compliance criteria and appropriate solutions for well-understood best practices.

For Azure Cognitive Search, there is currently one built-definition, listed below, that you can use in a policy assignment. The built-in is for logging and monitoring. By using this built-in definition in a policy that you create, the system will scan for search services that do not have resource logging, and then enable it accordingly.

Regulatory Compliance in Azure Policy provides Microsoft-created and managed initiative definitions, known as *built-ins*, for the **compliance domains** and **security controls** related to different compliance standards. This page lists the **compliance domains** and **security controls** for Azure Cognitive Search. You can assign the built-ins for a **security control** individually to help make your Azure resources compliant with the specific standard.

The title of each built-in policy definition links to the policy definition in the Azure portal. Use the link in the **Policy Version** column to view the source on the Azure Policy GitHub repo.

> **Important**
>
> Each control is associated with one or more **Azure Policy** definitions. These policies might help you **assess compliance** with the control. However, there often isn't a one-to-one or complete match between a control and one or more policies. As such, **Compliant** in Azure Policy refers only to the policies themselves. This doesn't ensure that you're fully compliant with all requirements of a control. In addition, the compliance standard includes controls that aren't addressed by any Azure Policy definitions at this time. Therefore, compliance in Azure Policy is only a partial view of your overall compliance status. The associations between controls and Azure Policy Regulatory Compliance definitions for these compliance standards can change over time.

Azure Security Benchmark

The Azure Security Benchmark provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the Azure Security Benchmark mapping files.

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Azure Security Benchmark.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Logging and Threat Detection	LT-3	Enable logging for security investigation	Resource logs in Search services should be enabled	5.0.0

Azure Security Benchmark v1

The Azure Security Benchmark provides recommendations on how you can secure your cloud solutions on Azure. To see how this service completely maps to the Azure Security Benchmark, see the Azure Security Benchmark mapping files.

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - Azure Security Benchmark.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Logging and Monitoring	2.3	Enable audit logging for Azure resources	Resource logs in Search services should be enabled	5.0.0

CIS Microsoft Azure Foundations Benchmark 1.3.0

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - CIS Microsoft Azure Foundations Benchmark 1.3.0. For more information about this compliance standard, see CIS Microsoft Azure Foundations Benchmark.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
5 Logging and Monitoring	CIS Microsoft Azure Foundations Benchmark recommendation 5.3	Ensure that Diagnostic Logs are enabled for all services which support it.	Resource logs in Search services should be enabled	5.0.0

CIS Microsoft Azure Foundations Benchmark 1.4.0

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance details for CIS v1.4.0. For more information about this compliance standard, see CIS Microsoft Azure Foundations Benchmark.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
5 Logging and Monitoring	CIS Microsoft Azure Foundations Benchmark recommendation 5.3	Ensure that Diagnostic Logs Are Enabled for All Services that Support it.	Resource logs in Search services should be enabled	5.0.0

FedRAMP High

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - FedRAMP High. For more information about this compliance standard, see FedRAMP High.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should disable public network access	1.0.0

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Cognitive Search services should use private link	1.0.0
Audit And Accountability	AU-6 (4)	Central Review And Analysis	Resource logs in Search services should be enabled	5.0.0
Audit And Accountability	AU-6 (5)	Integration / Scanning And Monitoring Capabilities	Resource logs in Search services should be enabled	5.0.0
Audit And Accountability	AU-12	Audit Generation	Resource logs in Search services should be enabled	5.0.0
Audit And Accountability	AU-12 (1)	System-Wide / Time-Correlated Audit Trail	Resource logs in Search services should be enabled	5.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should disable public network access	1.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should use private link	1.0.0
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search service should use a SKU that supports private link	1.0.0

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should disable public network access	1.0.0
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should use private link	1.0.0

FedRAMP Moderate

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - FedRAMP Moderate. For more information about this compliance standard, see FedRAMP Moderate.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should disable public network access	1.0.0
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17 (1)	Automated Monitoring / Control	Azure Cognitive Search services should use private link	1.0.0

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Audit And Accountability	AU-12	Audit Generation	Resource logs in Search services should be enabled	5.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should disable public network access	1.0.0
System And Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should use private link	1.0.0
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should disable public network access	1.0.0
System And Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should use private link	1.0.0

HIPAA HITRUST 9.2

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - HIPAA HITRUST 9.2. For more information about this compliance standard, see HIPAA HITRUST 9.2 .

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
12 Audit Logging & Monitoring	1208.09aa3System.1-09.aa	1208.09aa3System.1-09.aa 09.10 Monitoring	Resource logs in Search services should be enabled	5.0.0

New Zealand ISM Restricted

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - New Zealand ISM Restricted. For more information about this compliance standard, see New Zealand ISM Restricted.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control and Passwords	AC-17	16.6.9 Events to be logged	Resource logs in Search services should be enabled	5.0.0

NIST SP 800-53 Rev. 5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - NIST SP 800-53 Rev. 5. For more information about this compliance standard, see NIST SP 800-53 Rev. 5.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should disable public network access	1.0.0
Access Control	AC-4	Information Flow Enforcement	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17	Remote Access	Azure Cognitive Search services should use private link	1.0.0
Access Control	AC-17 (1)	Monitoring and Control	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
Access Control	AC-17 (1)	Monitoring and Control	Azure Cognitive Search services should use private link	1.0.0

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Audit and Accountability	AU-6 (4)	Central Review and Analysis	Resource logs in Search services should be enabled	5.0.0
Audit and Accountability	AU-6 (5)	Integrated Analysis of Audit Records	Resource logs in Search services should be enabled	5.0.0
Audit and Accountability	AU-12	Audit Record Generation	Resource logs in Search services should be enabled	5.0.0
Audit and Accountability	AU-12 (1)	System-wide and Time-correlated Audit Trail	Resource logs in Search services should be enabled	5.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should disable public network access	1.0.0
System and Communications Protection	SC-7	Boundary Protection	Azure Cognitive Search services should use private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search service should use a SKU that supports private link	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should disable public network access	1.0.0
System and Communications Protection	SC-7 (3)	Access Points	Azure Cognitive Search services should use private link	1.0.0

NZ ISM Restricted v3.5

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - NZ ISM Restricted v3.5. For more information about this compliance standard, see NZ ISM Restricted v3.5 .

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Access Control and Passwords	NZISM Security Benchmark AC-18	16.6.9 Events to be logged	Resource logs in Search services should be enabled	5.0.0

RMIT Malaysia

To review how the available Azure Policy built-ins for all Azure services map to this compliance standard, see Azure Policy Regulatory Compliance - RMIT Malaysia. For more information about this compliance standard, see RMIT Malaysia.

Domain	Control ID	Control title	Policy (Azure portal)	Policy version (GitHub)
Security of Digital Services	RMiT 10.66	Security of Digital Services - 10.66	Deploy Diagnostic Settings for Search Services to Event Hub	2.0.0
Security of Digital Services	RMiT 10.66	Security of Digital Services - 10.66	Deploy Diagnostic Settings for Search Services to Log Analytics workspace	1.0.0

Next steps

- Learn more about Azure Policy Regulatory Compliance.
- See the built-ins on the Azure Policy GitHub repo.

Additional resources

Documentation

Azure security baseline for Azure Cognitive Search

The Azure Cognitive Search security baseline provides procedural guidance and resources for implementing the security recommendations specified in the Microsoft cloud security benchmark.

Built-in policy definitions for Azure Cognitive Services - Azure Cognitive Services

Lists Azure Policy built-in policy definitions for Azure Cognitive Services. These built-in policy definitions provide common approaches to managing your Azure resources.

Security overview - Azure Cognitive Search

Learn about the security features in Azure Cognitive Search to protect endpoints, content, and operations.

Preview feature list - Azure Cognitive Search

Preview features are released so that customers can provide feedback on their design and utility. This article is a comprehensive list of all features currently in preview.

Encrypt data using customer-managed keys - Azure Cognitive Search

Supplement server-side encryption in Azure Cognitive Search using customer managed keys (CMK) or bring your own keys (BYOK) that you create and manage in Azure Key Vault.

Configure an IP firewall - Azure Cognitive Search

Configure IP control policies to restrict access to your Azure Cognitive Search service to specific IP addresses.

Manage with REST - Azure Cognitive Search

Create and configure an Azure Cognitive Search service with the Management REST API. The Management REST API is comprehensive in scope, with access to generally available and preview features.

Connect using Azure roles - Azure Cognitive Search

Use Azure role-based access control for granular permissions on service administration and content tasks.

Show 5 more

Training

Learning path

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Certification

Microsoft Certified: Azure Developer Associate - Certifications

Azure developers design, build, test, and maintain cloud applications and services.

Azure security baseline for Azure Cognitive Search

Article • 11/18/2022

This security baseline applies guidance from the Microsoft cloud security benchmark version 1.0 to Azure Cognitive Search. The Microsoft cloud security benchmark provides recommendations on how you can secure your cloud solutions on Azure. The content is grouped by the security controls defined by the Microsoft cloud security benchmark and the related guidance applicable to Azure Cognitive Search.

You can monitor this security baseline and its recommendations using Microsoft Defender for Cloud. Azure Policy definitions will be listed in the Regulatory Compliance section of the Microsoft Defender for Cloud dashboard.

When a feature has relevant Azure Policy Definitions, they are listed in this baseline to help you measure compliance to the Microsoft cloud security benchmark controls and recommendations. Some recommendations may require a paid Microsoft Defender plan to enable certain security scenarios.

> **Note**
>
> **Features** not applicable to Azure Cognitive Search have been excluded. To see how Azure Cognitive Search completely maps to the Microsoft cloud security benchmark, see the **full Azure Cognitive Search security baseline mapping file**.

Security profile

The security profile summarizes high-impact behaviors of Azure Cognitive Search, which may result in increased security considerations.

Service Behavior Attribute	Value
Product Category	AI+ML, Mobile, Web
Customer can access HOST / OS	No Access
Service can be deployed into customer's virtual network	False
Stores customer content at rest	True

Network security

For more information, see the Microsoft cloud security benchmark: Network security.

NS-1: Establish network segmentation boundaries

Features

Virtual Network Integration

Description: Service supports deployment into customer's private Virtual Network (VNet). Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Network Security Group Support

Description: Service network traffic respects Network Security Groups rule assignment on its subnets. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

NS-2: Secure cloud services with network controls

Features

Azure Private Link

Description: Service native IP filtering capability for filtering network traffic (not to be confused with NSG or Azure Firewall). Learn more.

Supported	Enabled By Default	Configuration Responsibility

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: For outbound connections through a private endpoint, refer to: Make outbound connections through a private endpoint

Configuration Guidance: Deploy private endpoints to establish a private access point for the resources. Block all connections on the public endpoint for your search service. Increase security for the virtual network, by enabling you to block exfiltration of data from the virtual network.

Reference: Create a Private Endpoint for a secure connection to Azure Cognitive Search

Disable Public Network Access

Description: Service supports disabling public network access either through using service-level IP ACL filtering rule (not NSG or Azure Firewall) or using a 'Disable Public Network Access' toggle switch. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Azure Cognitive Search supports IP rules for inbound access through a firewall, similar to the IP rules you'll find in an Azure virtual network security group. By leveraging IP rules, you can restrict search service access to an approved set of machines and cloud services. Access to data stored in your search service from the approved sets of machines and services will still require the caller to present a valid authorization token.

Reference: Configure an IP firewall for Azure Cognitive Search

Identity management

For more information, see the Microsoft cloud security benchmark: Identity management.

IM-1: Use centralized identity and authentication system

Features

Azure AD Authentication Required for Data Plane Access

Description: Service supports using Azure AD authentication for data plane access. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Local Authentication Methods for Data Plane Access

Description: Local authentications methods supported for data plane access, such as a local username and password. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Feature notes: Avoid the usage of local authentication methods or accounts, these should be disabled wherever possible. Instead use Azure AD to authenticate where possible.

Configuration Guidance: Cognitive Search uses key-based authentication as its primary authentication methodology. For inbound requests to a search service endpoint, such as requests that create or query an index, API keys are the only generally available authentication option you have.

Reference: Use API keys for Azure Cognitive Search authentication

IM-3: Manage application identities securely and automatically

Features

Managed Identities

Description: Data plane actions support authentication using managed identities. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Feature notes: Managed Identity is not supported to allow inbound connection to Azure Cognitive Search. However, outbound connections to other Azure Services using Managed Identity is supported. Refer to: Connect a search service to other Azure resources using a managed identity

Configuration Guidance: This feature is not supported to secure this service.

Service Principals

Description: Data plane supports authentication using service principals. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: For services that don't support managed identities, use Azure Active Directory (Azure AD) to create a service principal with restricted permissions at the resource level. Configure service principals with certificate credentials and fall back to client secrets for authentication.

IM-7: Restrict resource access based on conditions

Features

Conditional Access for Data Plane

Description: Data plane access can be controlled using Azure AD Conditional Access Policies. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Define the applicable conditions and criteria for Azure Active Directory (Azure AD) conditional access in the workload. Consider common use cases such as blocking or granting access from specific locations, blocking risky sign-in behavior, or requiring organization-managed devices for specific applications.

IM-8: Restrict the exposure of credential and secrets

Features

Service Credential and Secrets Support Integration and Storage in Azure Key Vault

Description: Data plane supports native use of Azure Key Vault for credential and secrets store. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Privileged access

For more information, see the Microsoft cloud security benchmark: Privileged access.

PA-1: Separate and limit highly privileged/administrative users

Features

Local Admin Accounts

Description: Service has the concept of a local administrative account. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

PA-7: Follow just enough administration (least privilege) principle

Features

Azure RBAC for Data Plane

Description: Azure Role-Based Access Control (Azure RBAC) can be used to managed access to service's data plane actions. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Azure provides a global role-based access control (RBAC) authorization system for all services running on the platform. In Cognitive Search, you can:

- Use generally available roles for service administration.
- Use new preview roles for data requests, including creating, loading, and querying indexes.

Reference: Use Azure role-based access controls (Azure RBAC) in Azure Cognitive Search

PA-8: Determine access process for cloud provider support

Features

Customer Lockbox

Description: Customer Lockbox can be used for Microsoft support access. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: In support scenarios where Microsoft needs to access your data, use Customer Lockbox to review, then approve or reject each of Microsoft's data access requests.

Data protection

For more information, see the Microsoft cloud security benchmark: Data protection.

DP-1: Discover, classify, and label sensitive data

Features

Sensitive Data Discovery and Classification

Description: Tools (such as Azure Purview or Azure Information Protection) can be used for data discovery and classification in the service. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

DP-2: Monitor anomalies and threats targeting sensitive data

Features

Data Leakage/Loss Prevention

Description: Service supports DLP solution to monitor sensitive data movement (in customer's content). Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

DP-3: Encrypt sensitive data in transit

Features

Data in Transit Encryption

Description: Service supports data in-transit encryption for data plane. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: Azure Cognitive Search data in transit encryption

DP-4: Enable data at rest encryption by default

Features

Data at Rest Encryption Using Platform Keys

Description: Data at-rest encryption using platform keys is supported, any customer content at rest is encrypted with these Microsoft managed keys. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	True	Microsoft

Configuration Guidance: No additional configurations are required as this is enabled on a default deployment.

Reference: Azure Cognitive Search default data encryption using service-managed keys

DP-5: Use customer-managed key option in data at rest encryption when required

Features

Data at Rest Encryption Using CMK

Description: Data at-rest encryption using customer-managed keys is supported for customer content stored by the service. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: If required for regulatory compliance, define the use case and service scope where encryption using customer-managed keys are needed. Enable and implement data at rest encryption using customer-managed key for those services.

Reference: Configure customer-managed keys for data encryption in Azure Cognitive Search

DP-6: Use a secure key management process

Features

Key Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer keys, secrets, or certificates. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Azure Key Vault to create and control the life cycle of your encryption keys, including key generation, distribution, and storage. Rotate and revoke your keys in Azure Key Vault and your service based on a defined schedule or when there is a key retirement or compromise. When there is a need to use customer-managed key (CMK) in the workload, service, or application level, ensure you follow the best practices for key management: Use a key hierarchy to generate a separate data encryption key (DEK) with your key encryption key (KEK) in your key vault. Ensure keys are registered with Azure Key Vault and referenced via key IDs from the service or application. If you need to bring your own key (BYOK) to the service (such as importing HSM-protected keys from your on-premises HSMs into Azure Key Vault), follow recommended guidelines to perform initial key generation and key transfer.

Reference: Configure customer-managed keys for data encryption in Azure Cognitive Search

DP-7: Use a secure certificate management process

Features

Certificate Management in Azure Key Vault

Description: The service supports Azure Key Vault integration for any customer certificates. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Asset management

For more information, see the Microsoft cloud security benchmark: Asset management.

AM-2: Use only approved services

Features

Azure Policy Support

Description: Service configurations can be monitored and enforced via Azure Policy. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Use Microsoft Defender for Cloud to configure Azure Policy to audit and enforce configurations of your Azure resources. Use Azure Monitor to create alerts when there is a configuration deviation detected on the resources. Use Azure Policy [deny] and [deploy if not exists] effects to enforce secure configuration across Azure resources.

Reference: Azure Cognitive Search Policies

Logging and threat detection

For more information, see the Microsoft cloud security benchmark: Logging and threat detection.

LT-1: Enable threat detection capabilities

Features

Microsoft Defender for Service / Product Offering

Description: Service has an offering-specific Microsoft Defender solution to monitor and alert on security issues. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

LT-4: Enable logging for security investigation

Features

Azure Resource Logs

Description: Service produces resource logs that can provide enhanced service-specific metrics and logging. The customer can configure these resource logs and send them to their own data sink like a storage account or log analytics workspace. Learn more.

Supported	Enabled By Default	Configuration Responsibility
True	False	Customer

Configuration Guidance: Enable resource logs for the service to view the Azure Cognitive Search operations logs, search metrics, and etc.

Reference: Azure Cognitive Search resource log

Backup and recovery

For more information, see the Microsoft cloud security benchmark: Backup and recovery.

BR-1: Ensure regular automated backups

Features

Azure Backup

Description: The service can be backed up by the Azure Backup service. Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Configuration Guidance: This feature is not supported to secure this service.

Service Native Backup Capability

Description: Service supports its own native backup capability (if not using Azure Backup). Learn more.

Supported	Enabled By Default	Configuration Responsibility
False	Not Applicable	Not Applicable

Feature notes: Because Azure Cognitive Search isn't a primary data storage solution, Microsoft doesn't provide a formal mechanism for self-service backup and restore. However, you can backup and restore the index using your own code. Refer to: Back up and restore alternatives

Configuration Guidance: This feature is not supported to secure this service.

Next steps

- See the Microsoft cloud security benchmark overview
- Learn more about Azure security baselines

Create an Azure Cognitive Search service in the portal

Article • 02/24/2023 • 9 minutes to read

Azure Cognitive Search is an Azure resource used for adding a full text search experience to custom apps.

If you have an Azure subscription, including a trial subscription, you can create a search service for free. Free services have limitations, but you can complete all of the quickstarts and most tutorials.

The easiest way to create search service is using the Azure portal, which is covered in this article. You can also use Azure PowerShell, Azure CLI, the Management REST API, an Azure Resource Manager service template, or a Bicep file.

Before you start

The following service properties are fixed for the lifetime of the service. Because they're fixed, consider the usage implications as you fill in each property:

- Service name becomes part of the URL endpoint (review tips for helpful service names).
- Tier (Free, Basic, Standard, and so forth) determines the underlying physical hardware and billing. Some features are tier-constrained.

- Service region can determine the availability of certain scenarios. If you need high availability or AI enrichment, you'll need to create the resource in a region that provides the feature.

Subscribe (free or paid)

To try search for free, open a free Azure account and then create your search service using the **Free** tier. Because the tier is fixed, it will never transition to become a billable tier.

Alternatively, you can use free credits to try out paid Azure services, which means you can create your search service at **Basic** or above to get more capacity. Your credit card is never charged unless you explicitly change your settings and ask to be charged. Another approach is to activate Azure credits in a Visual Studio subscription. A Visual Studio subscription gives you credits every month you can use for paid Azure services.

Paid (or billable) search occurs when you choose a billable tier (Basic or above) when creating the resource on a billable Azure subscription.

Find the Azure Cognitive Search offering

1. Sign in to the Azure portal.
2. Click the plus sign ("**+ Create Resource**") in the top-left corner.
3. Use the search bar to find "Azure Cognitive Search" or navigate to the resource through **Web** > **Azure Cognitive Search**.

Choose a subscription

If you've more than one subscription, choose one for your search service. If you're implementing customer-managed encryption or if you use other features that depend on managed service identities for external data access, choose the same subscription as the one used for Azure Key Vault or other services for which managed identities are used.

Set a resource group

A resource group is a container that holds related resources for your Azure solution. It's required for the search service. It's also useful for managing resources all-up, including costs. A resource group can consist of one service, or multiple services used together. For example, if you're using Azure Cognitive Search to index an Azure Cosmos DB database, you could make both services part of the same resource group for management purposes.

If you aren't combining resources into a single group, or if existing resource groups are filled with resources used in unrelated solutions, create a new resource group just for your Azure Cognitive Search resource.

Over time, you can track current and projected costs all-up or you can view charges for individual resources. The following screenshot shows the kind of cost information you can expect to see when you combine multiple resources into one group.

> **Tip**
>
> Resource groups simplify cleanup because deleting a group deletes all of the services within it. For prototype projects utilizing multiple services, putting all of them in the same resource group makes cleanup easier after the project is over.

Name the service

In Instance Details, provide a service name in the **URL** field. The name is part of the URL endpoint against which API calls are issued: `https://your-service-name.search.windows.net`. For example, if you want the endpoint to be `https://myservice.search.windows.net`, you would enter `myservice`.

Service name requirements:

- It must be unique within the search.windows.net namespace
- It must be between 2 and 60 characters in length
- You must use lowercase letters, digits, or dashes ("-")
- Don't use dashes ("-") in the first 2 characters or as the last single character
- You may not use consecutive dashes ("--") anywhere

> **Tip**
>
> If you think you'll be using multiple services, we recommend including the region (or location) in the service name as a naming convention. Services within the same region can exchange data at no charge, so if Azure Cognitive Search is in West US, and you have other services also in West US, a name like `mysearchservice-westus` can save you a trip to the properties page when deciding how to combine or attach resources.

Choose a region

Azure Cognitive Search is available in most regions, as listed in the **Products available by region** page.

As a rule, if you're using multiple Azure services, putting all of them in the same region minimizes or voids bandwidth charges. There are no charges for data exchanges among services when all of them are in the same region.

Two notable exceptions might lead to provisioning one or more search services in a separate region:

- Outbound connections from Cognitive Search to Azure Storage. You might want storage in a different region if you're enabling a firewall.
- Business continuity and disaster recovery (BCDR) requirements dictate creating multiple search services in regional pairs. For example, if you're operating in North America, you might choose East US and West US, or North Central US and South Central US, for each search service.

Some features are subject to regional availability. If you require any of following features, choose a region that provides them:

- AI enrichment requires Cognitive Services to be in the same physical region as Azure Cognitive Search. There are just a few regions that *don't* provide both. The Products available by region page indicates a common regional presence by showing two stacked check marks. An unavailable combination has a missing check mark. The time piece icon indicates future availability.

	AUSTRALIA		BRAZIL	CANADA	
Products	Australia East	Australia Southeast	Brazil South	Canada Central	Canada East
Azure Cognitive Search	✓	✓	✓	✓	✓
AI Enrichment	✓	🕒	✓	✓	
Semantic Search	☐				

- Semantic search is currently in preview in selected regions, such as "Australia East" in the above screenshot.

Other features that have regional constraints:

- Availability Zones
- Azure roles for data plane operations (Azure public cloud only)

Choose a tier

Azure Cognitive Search is currently offered in multiple pricing tiers: Free, Basic, Standard, or Storage Optimized. Each tier has its own capacity and limits. Also the tier you select may impact the availability of certain features. See Feature availability by tier for guidance.

Basic and Standard are the most common choices for production workloads, but initially many customers start with the Free service for evaluation purposes. Among the billable tiers, key differences are partition size and speed, and limits on the number of objects you can create.

Select Pricing Tier

Browse available skus and their features

Sku	Offering	Indexes	Indexers	Storage	Search units	Replicas	Partitions	Search unit cost/month (est.)
F	Free	3	3	50 MB	1	1	1	<est. pricing removed>
B	Basic	15	15	2 GB	3	3	1	
S	Standard	50	50	25 GB/Partition*	36	12	12	
S2	Standard	200	200	100 GB/Partition*	36	12	12	
S3	Standard	200	200	200 GB/Partition*	36	12	12	
S3HD	High-density	1000	0	200 GB/Partition*	36	12	3	
L1	Storage Optimized	10	10	1 TB/Partition*	36	12	12	
L2	Storage Optimized	10	10	2 TB/Partition*	36	12	12	

Select

Prices presented are estimates in your local currency that include only Azure infrastructure costs and any discounts for the subscription and location. The prices don't include any applicable software costs. Final charges will appear in your local currency in cost analysis and billing views. View Azure pricing calculator.

Remember, a pricing tier can't be changed once the service is created. If you need a higher or lower tier, you'll have to re-create the service.

Create your service

After you've provided the necessary inputs, go ahead and create the service.

Your service is deployed within minutes. You can monitor progress through Azure notifications. Consider pinning the service to your dashboard for easy access in the future.

Configure authentication

Unless you're using the portal, programmatic access to your new service requires that you provide the URL endpoint and an authenticated connection. You can use either or both of these options:

- Connect using key-based authentication
- Connect using Azure roles

1. When setting up a programmatic connection, you'll need the search service endpoint. On the **Overview** page, locate and copy the URL endpoint on the right side of the page.

2. To set authentication options, use the **Keys** page. Most quickstarts and tutorials use API keys for simplicity, but if you're setting up a service for production workloads, consider using Azure roles. You can copy keys from this page.

An endpoint and key aren't needed for portal-based tasks. The portal is already linked to your Azure Cognitive Search resource with admin rights. For a portal walkthrough, start with Quickstart: Create an Azure Cognitive Search index in the portal.

Scale your service

After a search service is provisioned, you can scale it to meet your needs. If you chose the Standard tier, you can scale the service in two dimensions: replicas and partitions. For the Basic tier, you can only add replicas. For the free service, scale isn't available.

Partitions allow your service to store and search through more documents.

Replicas allow your service to handle a higher load of search queries.

Adding resources increases your monthly bill. The pricing calculator can help you understand the billing ramifications of adding resources. Remember that you can adjust resources based on load. For example, you might increase resources to create a full initial index, and then reduce resources later to a level more appropriate for incremental indexing.

> **Important**
>
> A service must have **2 replicas for read-only SLA and 3 replicas for read/write SLA**.

1. Go to your search service page in the Azure portal.
2. In the left-navigation pane, select **Settings** > **Scale**.
3. Use the slidebar to add resources of either type.

When to add a second service

Most customers use just one service provisioned at a tier sufficient for expected load. One service can host multiple indexes, subject to the maximum limits of the tier you select, with each index isolated from another. In Azure Cognitive Search, requests can

only be directed to one index, minimizing the chance of accidental or intentional data retrieval from other indexes in the same service.

Although most customers use just one service, service redundancy might be necessary if operational requirements include the following:

- Business continuity and disaster recovery (BCDR). Azure Cognitive Search doesn't provide instant failover in the event of an outage.
- Multi-tenant architectures sometimes call for two or more services.
- Globally deployed applications might require search services in each geography to minimize latency.

> **Note**
>
> In Azure Cognitive Search, you cannot segregate indexing and querying operations; thus, you would never create multiple services for segregated workloads. An index is always queried on the service in which it was created (you cannot create an index in one service and copy it to another).

A second service isn't required for high availability. High availability for queries is achieved when you use 2 or more replicas in the same service. Replica updates are sequential, which means at least one is operational when a service update is rolled out. For more information about uptime, see Service Level Agreements.

Add more services to a subscription

Cognitive Search restricts the number of resources you can initially create in a subscription. If you exhaust your maximum limit, file a new support request to add more search services.

1. Sign in to the Azure portal and find your search service.
2. On the left-navigation pane, scroll down and select **New Support Request.**
3. In **Issue type**, choose **Service and subscription limits (quotas).**
4. Select the subscription that needs more quota.
5. Under **Quota type**, select **Search** and then select **Next**.
6. In the **Problem details** section, select **Enter details**.

7. Follow the prompts to select the location and tier for which you want to increase the limit.

8. Add the number of new services you would like to add to your quota. The value must not be empty and must between 0 to 100. For example, the maximum number of S2 services is 8. If you want 12 services, you would request 4 of S2 services.

9. When you're finished, select **Save and continue** to continue creating your support request.

10. Provide the additional information required to file the request, and then select **Next**.

11. On **Review + create**, review the details that you'll send to support, and then select **Create**.

Next steps

After provisioning a service, you can continue in the portal to create your first index.

Quickstart: Create an Azure Cognitive Search index in the portal

Want to optimize and save on your cloud spending?

Start analyzing costs with Cost Management

Additional resources

Documentation

Import data into a search index using Azure portal - Azure Cognitive Search

Learn about the Import Data wizard in the Azure portal used to create and load an index, and optionally invoke AI enrichment using built-in skills for natural language processing, translation, OCR, and image analysis.

Multi-language indexing for non-English search queries - Azure Cognitive Search

Create an index that supports multi-language content and then create queries scoped to that content.

Quickstart: Create a search index in the Azure portal - Azure Cognitive Search

Create, load, and query your first search index using the Import Data wizard in Azure portal. This quickstart uses a fictitious hotel dataset for sample data.

What's new in Azure Cognitive Search

Announcements of new and enhanced features, including a service rename of Azure Search to Azure Cognitive Search.

Quickstart: Create a skillset in the Azure portal - Azure Cognitive Search

In this portal quickstart, use the Import Data wizard to add cognitive skills to an indexing pipeline to generate searchable text from images and unstructured documents. Skills in this quickstart include OCR, image analysis, and natural language processing.

Quickstart: Create a demo app in Azure portal - Azure Cognitive Search

Run the Create demo app \wizard to generate HTML pages and script for an operational web app. The page includes a search bar, results area, sidebar, and typeahead support.

Index overview - Azure Cognitive Search

Explains what is a search index in Azure Cognitive Search and describes content, construction, physical expression, and the index schema.

Feature descriptions - Azure Cognitive Search

Explore the feature categories of Azure Cognitive Search.

Show 5 more

Training

Learning paths and modules

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Learning certificate

Microsoft Certified: Azure Fundamentals - Certifications

Azure Fundamentals validates foundational knowledge of cloud services and how those services are provided with Microsoft Azure.

Choose a pricing tier for Azure Cognitive Search

Article • 02/24/2023 • 5 minutes to read

Part of creating a search service means choosing a pricing tier (or SKU) that's fixed for the lifetime of the service. In the portal, tier is specified in the **Select Pricing Tier** page when you create the service. If you're provisioning through PowerShell or Azure CLI instead, the tier is specified through the `-Sku` parameter

The tier you select determines:

- Maximum number of indexes and other objects allowed on the service
- Size and speed of partitions (physical storage)
- Billable rate as a fixed monthly cost, but also an incremental cost if you add capacity

In a few instances, the tier you choose determines the availability of premium features.

Pricing - or the estimated monthly cost of running the service - are shown in the portal's **Select Pricing Tier** page. You should check service pricing to learn about estimated costs.

> **Note**
>
> Looking for information about "Azure SKUs"? Start with **Azure pricing** and then scroll down for links to per-service pricing pages.

Tier descriptions

Tiers include **Free**, **Basic**, **Standard**, and **Storage Optimized**. Standard and Storage Optimized are available with several configurations and capacities. The following screenshot from Azure portal shows the available tiers, minus pricing (which you can find in the portal and on the pricing page.

Select Pricing Tier

Browse available skus and their features

Sku	Offering	Indexes	Indexers	Storage	Search units	Replicas	Partitions	Cost/month (estimated)
F	Free	3	3	50 MB	1	1	1	
B	Basic	15	15	2 GB	3	3	1	**Estimated costs not included in this screenshot**
S	Standard	50	50	25 GB/Partition*	36	12	12	
S2	Standard	200	200	100 GB/Partition*	36	12	12	
S3	Standard	200	200	200 GB/Partition*	36	12	12	
S3HD	High-density	1000	0	200 GB/Partition*	36	12	3	
L1	Storage Optimized	10	10	1 TB/Partition*	36	12	12	
L2	Storage Optimized	10	10	2 TB/Partition*	36	12	12	

Free creates a limited search service for smaller projects, like running tutorials and code samples. Internally, system resources are shared among multiple subscribers. You cannot scale a free service or run significant workloads.

The most commonly used billable tiers include the following:

- **Basic** has just one partition but with the ability to meet SLA with its support for three replicas.
- **Standard (S1, S2, S3)** is the default. It gives you more flexibility in scaling for workloads. You can scale both partitions and replicas. With dedicated resources under your control, you can deploy larger projects, optimize performance, and increase capacity.

Some tiers are designed for certain types of work:

- **Standard 3 High Density (S3 HD)** is a *hosting mode* for S3, where the underlying hardware is optimized for a large number of smaller indexes and is intended for multitenancy scenarios. S3 HD has the same per-unit charge as S3, but the hardware is optimized for fast file reads on a large number of smaller indexes.
- **Storage Optimized (L1, L2)** tiers offer larger storage capacity at a lower price per TB than the Standard tiers. These tiers are designed for large indexes that don't change very often. The primary tradeoff is higher query latency, which you should validate for your specific application requirements.

You can find out more about the various tiers on the pricing page, in the Service limits in Azure Cognitive Search article, and on the portal page when you're provisioning a service.

Feature availability by tier

Most features are available on all tiers, including the free tier. In a few cases, the tier you choose will impact your ability to implement a feature. The following table describes

feature constraints that are related to service tier.

Feature	Limitations
indexers	Indexers are not available on S3 HD. Indexers have more limitations on the free tier.
AI enrichment	Runs on the Free tier but not recommended.
Managed or trusted identities for outbound (indexer) access	Not available on the Free tier.
Customer-managed encryption keys	Not available on the Free tier.
IP firewall access	Not available on the Free tier.
Private endpoint (integration with Azure Private Link)	For inbound connections to a search service, not available on the Free tier. For outbound connections by indexers to other Azure resources, not available on Free or S3 HD. For indexers that use skillsets, not available on Free, Basic, S1, or S3 HD.
Availability Zones	Not available on the Free or Basic tier.
Semantic search (preview)	Not available on the Free or Basic tier.

Resource-intensive features might not work well unless you give it sufficient capacity. For example, AI enrichment has long-running skills that time out on a Free service unless the dataset is small.

Upper limits

Tiers determine the maximum storage of the service itself, as well as the maximum number of indexes, indexers, data sources, skillsets, and synonym maps that you can create. For a full break out of all limits, see Service limits in Azure Cognitive Search.

Partition size and speed

Tier pricing includes details about per-partition storage that ranges from 2 GB for Basic, up to 2 TB for Storage Optimized (L2) tiers. Other hardware characteristics, such as speed of operations, latency, and transfer rates, are not published, but tiers that are designed for specific solution architectures are built on hardware that has the features

to support those scenarios. For more information about partitions, see Estimate and manage capacity and Reliability in Azure Cognitive Search.

Billing rates

Tiers have different billing rates, with higher rates for tiers that run on more expensive hardware or provide more expensive features. The per-tier billing rate can be found in the Azure pricing pages for Azure Cognitive Search.

Once you create a service, the billing rate becomes both a *fixed cost* of running the service around the clock, and an *incremental cost* if you choose to add more capacity.

Search services are allocated computing resources in the form of *partitions* (for storage), and *replicas* (instances of the query engine). Initially, a service is created with one of each, and the billing rate is inclusive of both resources. However, if you scale capacity, the costs go up or down in increments of the billable rate.

The following example provides an illustration. Assume a hypothetical billing rate of $100 per month. If you keep the search service at its initial capacity of one partition and one replica, then $100 is what you can expect to pay at the end of the month. However, if you add two more replicas to achieve high availability, the monthly bill increases to $300 ($100 for the first replica-partition pair, followed by $200 for the two replicas).

This billing model is based on the concept of applying the billing rate to the number *search units* (SU) used by a search service. All services are initially provisioned at one SU, but you can increase the SUs by adding either partitions or replicas to handle larger workloads. For more information, see How to estimate costs of a search service.

Next steps

The best way to choose a pricing tier is to start with a least-cost tier, and then allow experience and testing inform your decision to keep the service or create a new one at a higher tier. For next steps, we recommend that you create a search service at a tier that can accommodate the level of testing you propose to do, and then review the following guidance for recommendations on estimating cost and capacity.

- Create a search service
- Estimate costs
- Estimate capacity

Additional resources

Documentation

Plan and manage costs - Azure Cognitive Search

Learn about billable events, the billing model, and tips for cost control when running a Cognitive Search service.

Create a search service in the portal - Azure Cognitive Search

Learn how to set up an Azure Cognitive Search resource in the Azure portal. Choose resource groups, regions, and the SKU or pricing tier.

Availability and continuity - Azure Cognitive Search

learn how to make a search service highly available and resilient against period disruptions or even catastrophic failures.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Service limits for tiers and skus - Azure Cognitive Search

Service limits used for capacity planning and maximum limits on requests and responses for Azure Cognitive Search.

Security overview - Azure Cognitive Search

Learn about the security features in Azure Cognitive Search to protect endpoints, content, and operations.

Introduction to Azure Cognitive Search - Azure Cognitive Search

Azure Cognitive Search is a fully managed cloud search service from Microsoft. Learn about use cases, the development workflow, comparisons to other Microsoft search products, and how to get started.

Analyze performance - Azure Cognitive Search

TBD

Show 5 more

Training

Learning paths and modules

Optimize storage performance and costs using Azure Blob storage tiers - Training

Learn what the different storage tiers are for blob storage and how they can improve performance and reduce costs.

Service limits in Azure Cognitive Search

Article • 12/07/2022 • 12 minutes to read

Maximum limits on storage, workloads, and quantities of indexes and other objects depend on whether you provision Azure Cognitive Search at **Free**, **Basic**, **Standard**, or **Storage Optimized** pricing tiers.

- **Free** is a multi-tenant shared service that comes with your Azure subscription.
- **Basic** provides dedicated computing resources for production workloads at a smaller scale, but shares some networking infrastructure with other tenants.
- **Standard** runs on dedicated machines with more storage and processing capacity at every level. Standard comes in four levels: S1, S2, S3, and S3 HD. S3 High Density (S3 HD) is engineered for multi-tenancy and large quantities of small indexes (three thousand indexes per service). S3 HD doesn't provide the indexer feature and data ingestion must leverage APIs that push data from source to index.
- **Storage Optimized** runs on dedicated machines with more total storage, storage bandwidth, and memory than **Standard**. This tier targets large, slow-changing indexes. Storage Optimized comes in two levels: L1 and L2.

Subscription limits

You can create multiple services, limited only by the number of services allowed at each tier. For example, you could create up to 16 services at the Basic tier and another 16 services at the S1 tier within the same subscription. For more information about tiers, see Choose an SKU or tier for Azure Cognitive Search.

Maximum service limits can be raised upon request. If you need more services within the same subscription, file a support request.

Resource	Free[1]	Basic	S1	S2	S3	S3 HD	L1	L2
Maximum services	1	16	16	8	6	6	6	6
Maximum scale in search units (SU)[2]	N/A	3 SU	36 SU	36 SU	36 SU	36 SU	36 SU	36 SU

[1] Free is based on infrastructure that's shared with other customers. Because the hardware isn't dedicated, scale-up isn't supported on the free tier.

[2] Search units are billing units, allocated as either a *replica* or a *partition*. You need both resources for storage, indexing, and query operations. To learn more about SU computations, see Scale resource levels for query and index workloads.

Storage limits

A search service is constrained by disk space or by a hard limit on the maximum number of indexes or indexers, whichever comes first. The following table documents storage limits. For maximum object limits, see Limits by resource.

Resource	Free	Basic[1]	S1	S2	S3	S3 HD	L1	L2
Service level agreement (SLA)[2]	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Storage per partition	50 MB	2 GB	25 GB	100 GB	200 GB	200 GB	1 TB	2 TB
Partitions per service	N/A	1	12	12	12	3	12	12
Partition size	N/A	2 GB	25 GB	100 GB	200 GB	200 GB	1 TB	2 TB
Replicas	N/A	3	12	12	12	12	12	12

[1] Basic has one fixed partition. Additional search units can be used to add replicas for larger query volumes.

[2] Service level agreements are in effect for billable services on dedicated resources. Free services and preview features have no SLA. For billable services, SLAs take effect when you provision sufficient redundancy for your service. Two or more replicas are required for query (read) SLAs. Three or more replicas are required for query and indexing (read-write) SLAs. The number of partitions isn't an SLA consideration.

Index limits

Resource	Free	Basic [1]	S1	S2	S3	S3 HD	L1	L2
Maximum indexes	3	5 or 15	50	200	200	1000 per partition or 3000 per service	10	10
Maximum simple fields per index [2]	1000	100	1000	1000	1000	1000	1000	1000
Maximum complex collections per index	40	40	40	40	40	40	40	40
Maximum elements across all complex collections per document [3]	3000	3000	3000	3000	3000	3000	3000	3000
Maximum depth of complex fields	10	10	10	10	10	10	10	10
Maximum suggesters per index	1	1	1	1	1	1	1	1
Maximum scoring profiles per index	100	100	100	100	100	100	100	100

Resource	Free	Basic [1]	S1	S2	S3	S3 HD	L1	L2
Maximum functions per profile	8	8	8	8	8	8	8	8

[1] Basic services created before December 2017 have lower limits (5 instead of 15) on indexes. Basic tier is the only SKU with a lower limit of 100 fields per index.

[2] The upper limit on fields includes both first-level fields and nested subfields in a complex collection. For example, if an index contains 15 fields and has two complex collections with five subfields each, the field count of your index is 25. Indexes with a very large fields collection can be slow. Limit fields and attributes to just those you need, and run indexing and query test to ensure performance is acceptable.

[3] An upper limit exists for elements because having a large number of them significantly increases the storage required for your index. An element of a complex collection is defined as a member of that collection. For example, assume a Hotel document with a Rooms complex collection, each room in the Rooms collection is considered an element. During indexing, the indexing engine can safely process a maximum of 3000 elements across the document as a whole. This limit was introduced in `api-version=2019-05-06` and applies to complex collections only, and not to string collections or to complex fields.

You might find some variation in maximum limits if your service happens to be provisioned on a more powerful cluster. The limits here represent the common denominator. Indexes built to the above specifications will be portable across equivalent service tiers in any region.

Document limits

There are no longer any document limits per service in Azure Cognitive Search, however, there's a limit of approximately 24 billion documents per index on Basic, S1, S2, S3, L1, and L2 search services. For S3 HD, the limit is 2 billion documents per index. Each element of a complex collection counts as a separate document in terms of these limits.

Document size limits per API call

The maximum document size when calling an Index API is approximately 16 megabytes.

Document size is actually a limit on the size of the Index API request body. Since you can pass a batch of multiple documents to the Index API at once, the size limit realistically depends on how many documents are in the batch. For a batch with a single document, the maximum document size is 16 MB of JSON.

When estimating document size, remember to consider only those fields that can be consumed by a search service. Any binary or image data in source documents should be omitted from your calculations.

Indexer limits

Maximum running times exist to provide balance and stability to the service as a whole, but larger data sets might need more indexing time than the maximum allows. If an indexing job can't complete within the maximum time allowed, try running it on a schedule. The scheduler keeps track of indexing status. If a scheduled indexing job is interrupted for any reason, the indexer can pick up where it last left off at the next scheduled run.

Resource	Free [1]	Basic [2]	S1	S2	S3	S3 HD [3]	L1	L2
Maximum indexers	3	5 or 15	50	200	200	N/A	10	10
Maximum datasources	3	5 or 15	50	200	200	N/A	10	10
Maximum skillsets [4]	3	5 or 15	50	200	200	N/A	10	10
Maximum indexing load per invocation	10,000 documents	Limited only by maximum documents	Limited only by maximum documents	Limited only by maximum documents	Limited only by maximum documents	N/A	No limit	No limit
Minimum schedule	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes	5 minutes
Maximum running time [6]	1-3 minutes	2 or 24 hours	2 or 24 hours	2 or 24 hours	2 or 24 hours	N/A	2 or 24 hours	2 or 24 hours
Maximum running time for indexers with a skillset [5]	3-10 minutes	2 hours	2 hours	2 hours	2 hours	N/A	2 hours	2 hours
Blob indexer: maximum blob size, MB	16	16	128	256	256	N/A	256	256
Blob indexer: maximum characters of content extracted from a blob	32,000	64,000	4 million	8 million	16 million	N/A	4 million	4 million

[1] Free services have indexer maximum execution time of 3 minutes for blob sources and 1 minute for all other data sources. Indexer invocation is once every 180 seconds. For AI indexing that calls into Cognitive Services, free services are limited to 20 free transactions per indexer per day, where a transaction is defined as a document that successfully passes through the enrichment pipeline (tip: you can reset an indexer to reset its count).

[2] Basic services created before December 2017 have lower limits (5 instead of 15) on indexers, data sources, and skillsets.

[3] S3 HD services don't include indexer support.

[4] Maximum of 30 skills per skillset.

[5] AI enrichment and image analysis are computationally intensive and consume disproportionate amounts of available processing power. Running time for these workloads has been shortened to give other jobs in the queue more opportunity to run.

[6] Indexer execution and indexer-skillset combined execution is subject to a 2-hour maximum duration. Currently, some indexers have a longer 24-hour maximum execution window, but that behavior isn't the norm. The longer window only applies if a service or its indexers can't be internally migrated to the newer runtime behavior. If more than 2 hours are needed to complete an indexer or indexer-skillset process, schedule the indexer to run at 2-hour intervals.

> **Note**
>
> As stated in the **Index limits**, indexers will also enforce the upper limit of 3000 elements across all complex collections per document starting with the latest GA API version that supports complex types (`2019-05-06`) onwards. This means that if you've created your indexer with a prior API version, you will not be subject to this limit. To preserve maximum compatibility, an indexer that was created with a prior API version and then updated with an API version `2019-05-06` or later, will still be **excluded** from the limits. Customers should be aware of the adverse impact of having very large complex collections (as stated previously) and we highly recommend creating any new indexers with the latest GA API version.

Shared private link resource limits

Indexers can access other Azure resources over private endpoints managed via the shared private link resource API. This section describes the limits associated with this capability.

Resource	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Private endpoint indexer support	No	Yes	Yes	Yes	Yes	No	Yes	Yes

Resource	Free	Basic	S1	S2	S3	S3 HD	L1	L2
Private endpoint support for indexers with a skillset[1]	No	No	No	Yes	Yes	No	Yes	Yes
Maximum private endpoints	N/A	10 or 30	100	400	400	N/A	20	20
Maximum distinct resource types[2]	N/A	4	7	15	15	N/A	4	4

[1] AI enrichment and image analysis are computationally intensive and consume disproportionate amounts of available processing power. For this reason, private connections are disabled on lower tiers to avoid an adverse impact on the performance and stability of the search service itself.

[2] The number of distinct resource types are computed as the number of unique `groupId` values used across all shared private link resources for a given search service, irrespective of the status of the resource.

Synonym limits

Maximum number of synonym maps varies by tier. Each rule can have up to 20 expansions, where an expansion is an equivalent term. For example, given "cat", association with "kitty", "feline", and "felis" (the genus for cats) would count as 3 expansions.

Resource	Free	Basic	S1	S2	S3	S3-HD	L1	L2
Maximum synonym maps	3	3	5	10	20	20	10	10
Maximum number of rules per map	5000	20000	20000	20000	20000	20000	20000	20000

Index alias limits

Maximum number of index aliases varies by tier. In all tiers, the maximum number of aliases is double the maximum number of indexes allowed.

Resource	Free	Basic	S1	S2	S3	S3-HD	L1	L2
Maximum aliases	6	10 or 30	100	400	400	2000 per partition or 6000 per service	20	20

Data limits (AI enrichment)

An AI enrichment pipeline that makes calls to Azure Cognitive Services for Language resource for entity recognition, entity linking, key phrase extraction, sentiment analysis, language detection, and personal-information detection is subject to data limits. The maximum size of a

record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the sentiment analyzer, use the Text Split skill.

Throttling limits

API requests are throttled as the system approaches peak capacity. Throttling behaves differently for different APIs. Query APIs (Search/Suggest/Autocomplete) and indexing APIs throttle dynamically based on the load on the service. Index APIs and service operations API have static request rate limits.

Static rate request limits for operations related to an index:

- List Indexes (GET /indexes): 3 per second per search unit
- Get Index (GET /indexes/myindex): 10 per second per search unit
- Create Index (POST /indexes): 12 per minute per search unit
- Create or Update Index (PUT /indexes/myindex): 6 per second per search unit
- Delete Index (DELETE /indexes/myindex): 12 per minute per search unit

Static rate request limits for operations related to a service:

- Service Statistics (GET /servicestats): 4 per second per search unit

API request limits

- Maximum of 16 MB per request [1]
- Maximum 8 KB URL length
- Maximum 1000 documents per batch of index uploads, merges, or deletes
- Maximum 32 fields in $orderby clause
- Maximum 100,000 characters in a search clause
- The maximum number of clauses in `search` (expressions separated by AND or OR) is 1024
- Maximum search term size is 32,766 bytes (32 KB minus 2 bytes) of UTF-8 encoded text
- Maximum search term size is 1000 characters for prefix search and regex search
- Wildcard search and Regular expression search are limited to a maximum of 1000 states when processed by Lucene.

[1] In Azure Cognitive Search, the body of a request is subject to an upper limit of 16 MB, imposing a practical limit on the contents of individual fields or collections that aren't otherwise constrained by theoretical limits (see Supported data types for more information about field composition and restrictions).

Limits on query size and composition exist because unbounded queries can destabilize your search service. Typically, such queries are created programmatically. If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

API response limits

- Maximum 1000 documents returned per page of search results
- Maximum 100 suggestions returned per Suggest API request

API key limits

API keys are used for service authentication. There are two types. Admin keys are specified in the request header and grant full read-write access to the service. Query keys are read-only, specified on the URL, and typically distributed to client applications.

- Maximum of 2 admin keys per service
- Maximum of 50 query keys per service

Estimate and manage capacity of a search service

Article • 03/16/2023 • 16 minutes to read

Before you create a search service and lock in a specific pricing tier, take a few minutes to understand how capacity works and how you might adjust replicas and partitions to accommodate workload fluctuation.

In Azure Cognitive Search, capacity is based on *replicas* and *partitions* that can be scaled to your workload. Replicas are copies of the search engine. Partitions are units of storage. Each new search service starts with one each, but you can adjust each unit independently to accommodate fluctuating workloads. Adding either unit is billable.

The physical characteristics of replicas and partitions, such as processing speed and disk IO, vary by service tier. If you provisioned on Standard, replicas and partitions will be faster and larger than those of Basic.

Changing capacity isn't instantaneous. It can take up to an hour to commission or decommission partitions, especially on services with large amounts of data.

When scaling a search service, you can choose from the following tools and approaches:

- Azure portal
- Azure PowerShell
- Azure CLI
- Management REST API

Concepts: search units, replicas, partitions, shards

Capacity is expressed in *search units* that can be allocated in combinations of *partitions* and *replicas*, using an underlying *sharding* mechanism to support flexible configurations:

Concept	Definition
Search unit	A single increment of total available capacity (36 units). It's also the billing unit for an Azure Cognitive Search service. A minimum of one unit is required to run the service.
Replica	Instances of the search service, used primarily to load balance query operations. Each replica hosts one copy of an index. If you allocate three replicas, you'll have three copies of an index available for servicing query requests.

Concept	Definition
Partition	Physical storage and I/O for read/write operations (for example, when rebuilding or refreshing an index). Each partition has a slice of the total index. If you allocate three partitions, your index is divided into thirds.
Shard	A chunk of an index. Azure Cognitive Search divides each index into shards to make the process of adding partitions faster (by moving shards to new search units).

The following diagram shows the relationship between replicas, partitions, shards, and search units. It shows an example of how a single index is spanned across four search units in a service with two replicas and two partitions. Each of the four search units stores only half of the shards of the index. The search units in the left column store the first half of the shards, comprising the first partition, while those in the right column store the second half of the shards, comprising the second partition. Since there are two replicas, there are two copies of each index shard. The search units in the top row store one copy, comprising the first replica, while those in the bottom row store another copy, comprising the second replica.

The diagram above is only one example. Many combinations of partitions and replicas are possible, up to a maximum of 36 total search units.

In Cognitive Search, shard management is an implementation detail and nonconfigurable, but knowing that an index is sharded helps to understand the occasional anomalies in ranking and autocomplete behaviors:

- Ranking anomalies: Search scores are computed at the shard level first, and then aggregated up into a single result set. Depending on the characteristics of shard content, matches from one shard might be ranked higher than matches in another one. If you notice counter intuitive rankings in search results, it's most likely due to the effects of sharding, especially if indexes are small. You can avoid these ranking anomalies by choosing to compute scores globally across the entire index, but doing so will incur a performance penalty.

- Autocomplete anomalies: Autocomplete queries, where matches are made on the first several characters of a partially entered term, accept a fuzzy parameter that forgives small deviations in spelling. For autocomplete, fuzzy matching is constrained to terms within the current shard. For example, if a shard contains "Microsoft" and a partial term of "micor" is entered, the search engine will match on "Microsoft" in that shard, but not in other shards that hold the remaining parts of the index.

Approaching estimation

Capacity and the costs of running the service go hand in hand. Tiers impose limits on two levels: content (a count of indexes on a service, for example) and storage. It's important to consider both because whichever limit you reach first is the effective limit.

Counts of indexes and other objects are typically dictated by business and engineering requirements. For example, you might have multiple versions of the same index for active development, testing, and production.

Storage needs are determined by the size of the indexes you expect to build. There are no solid heuristics or generalities that help with estimates. The only way to determine the size of an index is build one. Its size will be based on imported data, text analysis, and index configuration such as whether you enable suggesters, filtering, and sorting.

For full text search, the primary data structure is an inverted index structure, which has different characteristics than source data. For an inverted index, size and complexity are determined by content, not necessarily by the amount of data that you feed into it. A large data source with high redundancy could result in a smaller index than a smaller dataset that contains highly variable content. So it's rarely possible to infer index size based on the size of the original dataset.

Attributes on the index, such as enabling filters and sorting, will impact storage requirements. The use of suggesters also has storage implications. For more information, see Attributes and index size.

> **Note**
>
> Even though estimating future needs for indexes and storage can feel like guesswork, it's worth doing. If a tier's capacity turns out to be too low, you'll need to provision a new service at a higher tier and then **reload your indexes**. There's no in-place upgrade of a service from one tier to another.

Estimate with the Free tier

One approach for estimating capacity is to start with the Free tier. Remember that the Free service offers up to three indexes, 50 MB of storage, and 2 minutes of indexing time. It can be challenging to estimate a projected index size with these constraints, but these are the steps:

- Create a free service.
- Prepare a small, representative dataset.
- Create an index and load your data. If the dataset can be hosted in an Azure data source supported by indexers, you can use the Import data wizard in the portal to both create and load the index. Otherwise, you could use REST and Postman to create the index and push the data. The push model requires data to be in the form of JSON documents, where fields in the document correspond to fields in the index.
- Collect information about the index, such as size. Features and attributes have an impact on storage. For example, adding suggesters (search-as-you-type queries) will increase storage requirements.

  Using the same data set, you might try creating multiple versions of an index, with different attributes on each field, to see how storage requirements vary. For more information, see "Storage implications" in Create a basic index.

With a rough estimate in hand, you might double that amount to budget for two indexes (development and production) and then choose your tier accordingly.

Estimate with a billable tier

Dedicated resources can accommodate larger sampling and processing times for more realistic estimates of index quantity, size, and query volumes during development. Some customers jump right in with a billable tier and then re-evaluate as the development project matures.

1. Review service limits at each tier to determine whether lower tiers can support the number of indexes you need. Across the Basic, S1, and S2 tiers, index limits are 15, 50, and 200, respectively. The Storage Optimized tier has a limit of 10 indexes because it's designed to support a low number of very large indexes.
2. Create a service at a billable tier:
   - Start low, at Basic or S1, if you're not sure about the projected load.

- Start high, at S2 or even S3, if testing includes large-scale indexing and query loads.
- Start with Storage Optimized, at L1 or L2, if you're indexing a large amount of data and query load is relatively low, as with an internal business application.

3. Build an initial index to determine how source data translates to an index. This is the only way to estimate index size.

4. Monitor storage, service limits, query volume, and latency in the portal. The portal shows you queries per second, throttled queries, and search latency. All of these values can help you decide if you selected the right tier.

5. Add replicas if you need high availability or if you experience slow query performance.

   There are no guidelines on how many replicas are needed to accommodate query loads. Query performance depends on the complexity of the query and competing workloads. Although adding replicas clearly results in better performance, the result isn't strictly linear: adding three replicas doesn't guarantee triple throughput. For guidance in estimating QPS for your solution, see Analyze performanceand Monitor queries.

> ⓘ **Note**
>
> Storage requirements can be inflated if you include data that will never be searched. Ideally, documents contain only the data that you need for the search experience. Binary data isn't searchable and should be stored separately (maybe in an Azure table or blob storage). A field should then be added in the index to hold a URL reference to the external data. The maximum size of an individual search document is 16 MB (or less if you're bulk uploading multiple documents in one request). For more information, see **Service limits in Azure Cognitive Search**.

Query volume considerations

Queries per second (QPS) is an important metric during performance tuning, but it's generally only a tier consideration if you expect high query volume at the outset.

The Standard tiers can provide a balance of replicas and partitions. You can increase query turnaround by adding replicas for load balancing or add partitions for parallel processing. You can then tune for performance after the service is provisioned.

If you expect high sustained query volumes from the outset, you should consider higher Standard tiers, backed by more powerful hardware. You can then take partitions and

replicas offline, or even switch to a lower-tier service, if those query volumes don't occur. For more information on how to calculate query throughput, see Monitor queries.

The Storage Optimized tiers are useful for large data workloads, supporting more overall available index storage for when query latency requirements are less important. You should still use additional replicas for load balancing and additional partitions for parallel processing. You can then tune for performance after the service is provisioned.

Service-level agreements

The Free tier and preview features aren't covered by service-level agreements (SLAs). For all billable tiers, SLAs take effect when you provision sufficient redundancy for your service. You need to have two or more replicas for query (read) SLAs. You need to have three or more replicas for query and indexing (read-write) SLAs. The number of partitions doesn't affect SLAs.

Tips for capacity planning

- Allow metrics to build around queries, and collect data around usage patterns (queries during business hours, indexing during off-peak hours). Use this data to inform service provisioning decisions. Though it's not practical at an hourly or daily cadence, you can dynamically adjust partitions and resources to accommodate planned changes in query volumes. You can also accommodate unplanned but sustained changes if levels hold long enough to warrant taking action.

- Remember that the only downside of under provisioning is that you might have to tear down a service if actual requirements are greater than your predictions. To avoid service disruption, you would create a new service at a higher tier and run it side by side until all apps and requests target the new endpoint.

When to add capacity

Initially, a service is allocated a minimal level of resources consisting of one partition and one replica. The tier you choose determines partition size and speed, and each tier is optimized around a set of characteristics that fit various scenarios. If you choose a higher-end tier, you might need fewer partitions than if you go with S1. One of the questions you'll need to answer through self-directed testing is whether a larger and more expensive partition yields better performance than two cheaper partitions on a service provisioned at a lower tier.

A single service must have sufficient resources to handle all workloads (indexing and queries). Neither workload runs in the background. You can schedule indexing for times

when query requests are naturally less frequent, but the service won't otherwise prioritize one task over another. Additionally, a certain amount of redundancy smooths out query performance when services or nodes are updated internally.

Some guidelines for determining whether to add capacity include:

- Meeting the high availability criteria for service level agreement
- The frequency of HTTP 503 errors is increasing
- Large query volumes are expected

As a general rule, search applications tend to need more replicas than partitions, particularly when the service operations are biased toward query workloads. Each replica is a copy of your index, allowing the service to load balance requests against multiple copies. All load balancing and replication of an index is managed by Azure Cognitive Search and you can alter the number of replicas allocated for your service at any time. You can allocate up to 12 replicas in a Standard search service and 3 replicas in a Basic search service. Replica allocation can be made either from the Azure portal or one of the programmatic options.

Search applications that require near real-time data refresh will need proportionally more partitions than replicas. Adding partitions spreads read/write operations across a larger number of compute resources. It also gives you more disk space for storing additional indexes and documents.

Finally, larger indexes take longer to query. As such, you might find that every incremental increase in partitions requires a smaller but proportional increase in replicas. The complexity of your queries and query volume will factor into how quickly query execution is turned around.

> **Note**
>
> Adding more replicas or partitions increases the cost of running the service, and can introduce slight variations in how results are ordered. Be sure to check the **pricing calculator** to understand the billing implications of adding more nodes. The **chart below** can help you cross-reference the number of search units required for a specific configuration. For more information on how additional replicas impact query processing, see **Ordering results**.

Add or reduce replicas and partitions

1. Sign in to the Azure portal and select the search service.

2. Under **Settings**, open the **Scale** page to modify replicas and partitions.

   The following screenshot shows a Basic Standard provisioned with one replica and partition. The formula at the bottom indicates how many search units are being used (1). If the unit price was $100 (not a real price), the monthly cost of running this service would be $100 on average.

3. Use the slider to increase or decrease the number of partitions. Select **Save**.

   This example adds a second replica and partition. Notice the search unit count; it's now four because the billing formula is replicas multiplied by partitions (2 x 2). Doubling capacity more than doubles the cost of running the service. If the search unit cost was $100, the new monthly bill would now be $400.

   For the current per unit costs of each tier, visit the Pricing page.

4. After saving, you can check notifications to confirm the action succeeded.

Changes in capacity can take anywhere from 15 minutes up to several hours to complete. You can't cancel once the process has started and there's no real-time monitoring for replica and partition adjustments. However, the following message remains visible while changes are underway.

> **Note**
>
> After a service is provisioned, it cannot be upgraded to a higher tier. You must create a search service at the new tier and reload your indexes. See **Create an Azure Cognitive Search service in the portal** for help with service provisioning.

How scale requests are handled

Upon receipt of a scale request, the search service:

1. Checks whether the request is valid.
2. Starts backing up data and system information.
3. Checks whether the service is already in a provisioning state (currently adding or eliminating either replicas or partitions).
4. Starts provisioning.

Scaling a service can take as little as 15 minutes or well over an hour, depending on the size of the service and the scope of the request. Backup can take several minutes, depending on the amount of data and number of partitions and replicas.

The above steps aren't entirely consecutive. For example, the system starts provisioning when it can safely do so, which could be while backup is winding down.

Errors during scaling

The error message "Service update operations aren't allowed at this time because we're processing a previous request" is caused by repeating a request to scale down or up when the service is already processing a previous request.

Resolve this error by checking service status to verify provisioning status:

1. Use the Management REST API, Azure PowerShell, or Azure CLI to get service status.
2. Call Get Service (REST) or equivalent for PowerShell or the CLI.
3. Check the response for "provisioningState": "provisioning"

If status is "Provisioning", wait for the request to complete. Status should be either "Succeeded" or "Failed" before another request is attempted. There's no status for backup. Backup is an internal operation and it's unlikely to be a factor in any disruption of a scale exercise.

If your search service appears to be stalled in a provisioning state, check for orphaned indexes that are unusable, with zero query volumes and no index updates. An unusable index can block changes to service capacity. In particular, look for indexes that are CMK-encrypted, whose keys are no longer valid. You should either delete the index or restore the keys to bring the index back online and unblock your scale operation.

Partition and replica combinations

A Basic service can have exactly one partition and up to three replicas, for a maximum limit of three SUs. The only adjustable resource is replicas. You need a minimum of two replicas for high availability on queries.

All Standard and Storage Optimized search services can assume the following combinations of replicas and partitions, subject to the 36-SU limit allowed for these tiers.

	1 partition	2 partitions	3 partitions	4 partitions	6 partitions	12 partitions
1 replica	1 SU	2 SU	3 SU	4 SU	6 SU	12 SU
2 replicas	2 SU	4 SU	6 SU	8 SU	12 SU	24 SU

	1 partition	2 partitions	3 partitions	4 partitions	6 partitions	12 partitions
3 replicas	3 SU	6 SU	9 SU	12 SU	18 SU	36 SU
4 replicas	4 SU	8 SU	12 SU	16 SU	24 SU	N/A
5 replicas	5 SU	10 SU	15 SU	20 SU	30 SU	N/A
6 replicas	6 SU	12 SU	18 SU	24 SU	36 SU	N/A
12 replicas	12 SU	24 SU	36 SU	N/A	N/A	N/A

SUs, pricing, and capacity are explained in detail on the Azure website. For more information, see Pricing Details.

> **Note**
>
> The number of replicas and partitions divides evenly into 12 (specifically, 1, 2, 3, 4, 6, 12). Azure Cognitive Search pre-divides each index into 12 shards so that it can be spread in equal portions across all partitions. For example, if your service has three partitions and you create an index, each partition will contain four shards of the index. How Azure Cognitive Search shards an index is an implementation detail, subject to change in future releases. Although the number is 12 today, you shouldn't expect that number to always be 12 in the future.

Next steps

Manage costs

Plan and manage costs of an Azure Cognitive Search service

Article • 12/01/2022 • 5 minutes to read

This article explains the billing model and billable events of Azure Cognitive Search, and provides direction for managing the costs.

As a first step, estimate your baseline costs by using the Azure pricing calculator. Alternatively, estimated costs and tier comparisons can also be found in the Select a pricing tier page when creating a service.

Azure provides built-in cost management that cuts across service boundaries to provide inclusive cost monitoring and the ability to set budgets and define alerts. The costs of running a search service will vary depending on capacity and which features you use. After you create your search service, optimize capacity so that you pay only for what you need.

Understand the billing model

Azure Cognitive Search runs on Azure infrastructure that accrues costs when you deploy new resources. It's important to understand that there could be other additional infrastructure costs that might accrue.

How you're charged for Azure Cognitive Search

When you create or use Search resources, you're charged for the following meters:

- You're charged an hourly rate based on the pricing tier of your search service, prorated to the hour.
- The charge is applied per the number of search units (SU) allocated to the service. Search units are units of capacity. Total SU is the product of replicas and partitions (R x P = SU) used by your service.

Billing is based on capacity (SUs) and the costs of running premium features, such as AI enrichment, Semantic search, and Private endpoints. Meters associated with premium features are listed in the following table.

Meter	Unit

Meter	Unit
Image extraction (AI enrichment) [1, 2]	Per 1000 images. See the pricing page.
Custom Entity Lookup skill (AI enrichment) [1]	Per 1000 text records. See the pricing page
Built-in skills (AI enrichment) [1]	Number of transactions, billed at the same rate as if you had performed the task by calling Cognitive Services directly. You can process 20 documents per indexer per day for free. Larger or more frequent workloads require a multi-resource Cognitive Services key.
Semantic Search [1]	Number of queries of "queryType=semantic", billed at a progressive rate. See the pricing page.
Private Endpoints [1]	Billed as long as the endpoint exists, and billed for bandwidth.

[1] Applies only if you use or enable the feature.

[2] In an indexer configuration, "imageAction" is the parameter that triggers image extraction. If "imageAction" is set to "none" (the default), you won't be charged for image extraction.

There is no meter on the number of queries, query responses, or documents ingested, although service limits do apply at each tier.

Data traffic might also incur networking costs. See the Bandwidth pricing.

Several premium features such as knowledge store, Debug Sessions, and enrichment cache have a dependency on Azure Storage. The meters for Azure Storage apply in this case, and the associated storage costs of using these features will be included in the Azure Storage bill.

Customer-managed keys provide double encryption of sensitive content. This feature requires a billable Azure Key Vault).

Skillsets can include billable built-in skills, non-billable built-in utility skills, and custom skills. Non-billable utility skills include Conditional, Shaper, Text Merge, Text Split. There

is no billing impact when using them, no Cognitive Services key requirement, and no 20 document limit.

A custom skill is functionality you provide. The cost of using a custom skill depends entirely on whether custom code is calling other metered services. There is no Cognitive Services key requirement and no 20 document limit on custom skills.

Monitor costs

Cost management is built into the Azure infrastructure. Review Billing and cost management for more information about tracking costs, tools, and APIs.

Minimize costs

Follow these guidelines to minimize costs of an Azure Cognitive Search solution.

1. If possible, create all resources in the same region, or in as few regions as possible, to minimize or eliminate bandwidth charges.

2. Scale up for resource-intensive operations like indexing, and then readjust downwards for regular query workloads. If there are predictable patterns to your workloads, you might be able to synchronize scale up to coincide with the expected volume (you would need to write code to automate this).

   When estimating the cost of a search solution, keep in mind that pricing and capacity aren't linear (doubling capacity more than doubles the cost on the same tier). Also, at some point, switching up to a higher tier can give you better and faster performance at roughly the same price point. For more information and an example, see Upgrade to a Standard S2 tier.

3. Consider Azure Web App for your front-end application so that requests and responses stay within the data center boundary.

4. If you're using AI enrichment, there is an extra charge for blob storage, but the cumulative cost goes down if you enable enrichment caching.

Create budgets

You can create budgets to manage costs and create alerts that automatically notify stakeholders of spending anomalies and overspending risks. Alerts are based on spending compared to budget and cost thresholds. Budgets and alerts are created for

Azure subscriptions and resource groups, so they're useful as part of an overall cost monitoring strategy.

Budgets can be created with filters for specific resources or services in Azure if you want more granularity present in your monitoring. Filters help ensure that you don't accidentally create new resources that cost you extra money. For more information about the filter options available when you create a budget, see Group and filter options.

Export cost data

You can also export your cost data to a storage account. This is helpful when you need or others to do more data analysis for costs. For example, a finance team can analyze the data using Excel or Power BI. You can export your costs on a daily, weekly, or monthly schedule and set a custom date range. Exporting cost data is the recommended way to retrieve cost datasets.

FAQ

Can I temporarily shut down a search service to save on costs?

Search runs as a continuous service. Dedicated resources are always operational, allocated for your exclusive use for the lifetime of your service. To stop billing entirely, you must delete the service. Deleting a service is permanent and also deletes its associated data.

Can I change the billing rate (tier) of an existing search service?

In-place upgrade or downgrade is not supported. Changing a service tier requires provisioning a new service at the desired tier.

Next steps

- Learn more on how pricing works with Azure Cognitive Search. See Azure Cognitive Search pricing page.
- Learn more about replicas and partitions.
- Learn how to optimize your cloud investment with Azure Cost Management.
- Learn more about managing costs with cost analysis.
- Learn about how to prevent unexpected costs.
- Take the Cost Management guided learning course.

Additional resources

Documentation

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Import data into a search index using Azure portal - Azure Cognitive Search

Learn about the Import Data wizard in the Azure portal used to create and load an index, and optionally invoke AI enrichment using built-in skills for natural language processing, translation, OCR, and image analysis.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Analyze performance - Azure Cognitive Search

TBD

Service limits for tiers and skus - Azure Cognitive Search

Service limits used for capacity planning and maximum limits on requests and responses for Azure Cognitive Search.

Index large data sets for full text search - Azure Cognitive Search

Strategies for large data indexing or computationally intensive indexing through batch mode, resourcing, and scheduled, parallel, and distributed indexing.

Data import and data ingestion - Azure Cognitive Search

Populate and upload data to an index in Azure Cognitive Search from external data sources.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Show 5 more

Training

Learning path

Control Azure spending and manage bills with Microsoft Cost Management and billing learning path - Training

Learn how to monitor and control your Azure spending and optimize the use of Azure resources.

Certification

Microsoft Certified: Azure AI Fundamentals - Certifications

Azure AI Fundamentals validates foundational knowledge of machine learning and artificial intelligence concepts and related Microsoft Azure services.

Reliability in Azure Cognitive Search

Article • 04/10/2023

Across Azure, reliability means maintaining resiliency and availability if there's a service outage or degradation. In Cognitive Search, reliability is achieved when you:

- Configure a service to use multiple replicas, paired with availability zone support.
- Deploy multiple services across different geographic regions.

All search workloads are fully contained within a single service that runs in a single geographic region. On a service, you can configure multiple replicas that automatically run in different availability zones. This capability is how you achieve high availability.

For business continuity and recovery from disasters at a regional level, you should develop a strategy that includes a cross-regional topology, consisting of multiple search services having identical configuration and content. Your custom script or code provides the "fail over" mechanism to an alternate search service if one suddenly becomes unavailable.

High availability

In Cognitive Search, replicas are copies of your index. A search service is installed with at least one replica, and can have up to 12 replicas. Adding replicas allows Azure Cognitive Search to do machine reboots and maintenance against one replica, while query execution continues on other replicas.

For each individual search service, Microsoft guarantees at least 99.9% availability for configurations that meet these criteria:

- Two replicas for high availability of read-only workloads (queries)
- Three or more replicas for high availability of read-write workloads (queries and indexing)

No SLA is provided for the Free tier. For more information, see SLA for Azure Cognitive Search.

Availability zone support

Availability zones are an Azure platform capability that divides a region's data centers into distinct physical location groups to provide high-availability, within the same

region. If you use availability zones for Cognitive Search, individual replicas are the units for zone assignment. A search service runs within one region; its replicas run in different zones.

You can utilize availability zones with Azure Cognitive Search by adding two or more replicas to your search service. Each replica is placed in a different availability zone within the region. If you have more replicas than availability zones, the replicas are distributed across availability zones as evenly as possible. There's no specific action on your part, except to create a search service in a region that provides availability zones, and then to configure the service to use multiple replicas.

Prerequisites

As noted, you must have multiple replicas for high availability: two for read-only query workloads, three for read-write workloads that include indexing.

Your service must be deployed in a region that supports availability zones. Azure Cognitive Search currently supports availability zones for Standard tier or higher, in one of the following regions:

Region	Roll out
Australia East	January 30, 2021 or later
Brazil South	May 2, 2021 or later
Canada Central	January 30, 2021 or later
Central India	January 20, 2022 or later
Central US	December 4, 2020 or later
China North 3	September 7, 2022 or later
East Asia	January 13, 2022 or later
East US	January 27, 2021 or later
East US 2	January 30, 2021 or later
France Central	October 23, 2020 or later
Germany West Central	May 3, 2021, or later
Japan East	January 30, 2021 or later
Korea Central	January 20, 2022 or later
North Europe	January 28, 2021 or later
Norway East	January 20, 2022 or later
Qatar Central	August 25, 2022 or later
South Africa North	September 7, 2022 or later
South Central US	April 30, 2021 or later
South East Asia	January 31, 2021 or later
Sweden Central	January 21, 2022 or later

Region	Roll out
Switzerland North	September 7, 2022 or later
UAE North	September 9, 2022 or later
UK South	January 30, 2021 or later
US Gov Virginia	April 30, 2021 or later
West Europe	January 29, 2021 or later
West US 2	January 30, 2021 or later
West US 3	June 02, 2021 or later

Availability Zones don't impact the Azure Cognitive Search Service Level Agreement. You still need three or more replicas for query high availability.

Multiple services in separate geographic regions

Service redundancy is necessary if operational requirements include:

- Business continuity and disaster recovery (BCDR) (Cognitive Search doesn't provide instant failover in the event of an outage).
- Global availability. If query and indexing requests come from all over the world, users who are closest to the host data center will have faster performance. Creating additional services in regions with close proximity to these users can equalize performance for all users.

If you need two or more search services, creating them in different regions can meet application requirements for continuity and recovery, as well as faster response times for a global user base.

Azure Cognitive Search doesn't provide an automated method of replicating search indexes across geographic regions, but there are some techniques that can make this process simple to implement and manage. These techniques are outlined in the next few sections.

The goal of a geo-distributed set of search services is to have two or more indexes available in two or more regions, where a user is routed to the Azure Cognitive Search service that provides the lowest latency:

You can implement this architecture by creating multiple services and designing a strategy for data synchronization. Optionally, you can include a resource like Azure Traffic Manager for routing requests.

Tip

For help in deploying multiple search services across multiple regions, see this Bicep sample on GitHub that deploys a fully configured, multi-regional search solution. The sample gives you two options for index synchronization, and request redirection using Traffic Manager.

Synchronize data across multiple services

There are two options for keeping two or more distributed search services in sync:

- Pull content updates into a search index by using an indexer.
- Push content into an index using the Add or Update Documents (REST) API or an Azure SDK equivalent API.

Option 1: Use indexers for updating content on multiple services

If you're already using indexer on one service, you can configure a second indexer on a second service to use the same data source object, pulling data from the same location. Each service in each region has its own indexer and a target index (your search index isn't shared, which means data is duplicated), but each indexer references the same data source.

Here's a high-level visual of what that architecture would look like.

Option 2: Use REST APIs for pushing content updates on multiple services

If you're using the Azure Cognitive Search REST API to push content to your search index, you can keep your various search services in sync by pushing changes to all search services whenever an update is required. In your code, make sure to handle cases where an update to one search service fails but succeeds for other search services.

Use Azure Traffic Manager to coordinate requests

Azure Traffic Manager allows you to route requests to multiple geo-located websites that are then backed by multiple search services. One advantage of the Traffic Manager is that it can probe Azure Cognitive Search to confirm availability and route users to alternate search services if a service is down. In addition, if you're routing search requests through Azure Web Sites, Azure Traffic Manager can help you load balance cases where the web site is up, but search isn't. Here's an example of an architecture that uses Traffic Manager.

Data residency

When you deploy multiple search services in different geographic regions, your content is stored in your chosen region.

Azure Cognitive Search won't store data outside of your specified region without your authorization. Specifically, the following features write to an Azure Storage resource: enrichment cache, debug session, knowledge store. The storage account is one that you provide, and it could be in any region.

If both the storage account and the search service are in the same region, network traffic between search and storage uses a private IP address and occurs over the Microsoft backbone network. Because private IP addresses are used, you can't configure IP firewalls or a private endpoint for network security. Instead, use the trusted service exception as an alternative when both services are in the same region.

Disaster recovery and service outages

As stated in the Service Level Agreement (SLA), Microsoft guarantees a high level of availability for index query requests when an Azure Cognitive Search service instance is configured with two or more replicas, and index update requests when an Azure

Cognitive Search service instance is configured with three or more replicas. However, there's no built-in mechanism for disaster recovery. If continuous service is required in the event of a catastrophic failure outside of Microsoft's control, we recommend provisioning a second service in a different region and implementing a geo-replication strategy to ensure indexes are fully redundant across all services.

Customers who use indexers to populate and refresh indexes can handle disaster recovery through geo-specific indexers that retrieve data from the same data source. Two services in different regions, each running an indexer, could index the same data source to achieve geo-redundancy. If you're indexing from data sources that are also geo-redundant, be aware that Azure Cognitive Search indexers can only perform incremental indexing (merging updates from new, modified, or deleted documents) from primary replicas. In a failover event, be sure to redirect the indexer to the new primary replica.

If you don't use indexers, you would use your application code to push objects and data to different search services in parallel. For more information, see Keep data synchronized across multiple services.

Back up and restore alternatives

Because Azure Cognitive Search isn't a primary data storage solution, Microsoft doesn't provide a formal mechanism for self-service backup and restore. However, you can use the **index-backup-restore** sample code in this Azure Cognitive Search .NET sample repo to back up your index definition and snapshot to a series of JSON files, and then use these files to restore the index, if needed. This tool can also move indexes between service tiers.

Otherwise, your application code used for creating and populating an index is the de facto restore option if you delete an index by mistake. To rebuild an index, you would delete it (assuming it exists), recreate the index in the service, and reload by retrieving data from your primary data store.

Next steps

- Review Service limits to learn more about the pricing tiers and services limits for each one.
- Review Plan for capacity to learn more about partition and replica combinations.
- Review Case Study: Use Cognitive Search to Support Complex AI Scenarios for real-world tips.

Migrate Azure Cognitive Search to availability zone support

Article • 02/24/2023 • 2 minutes to read

This guide describes how to migrate Azure Cognitive Search from non-availability zone support to availability support.

Azure Cognitive Search services can take advantage of availability support in regions that support availability zones. Services with two or more replicas in these regions created after availability support was enabled can automatically utilize availability zones. Each replica will be placed in a different availability zone within the region. If you have more replicas than availability zones, the replicas will be distributed across availability zones as evenly as possible.

If a search service was created before availability zone support was enabled in its region, the search service must be recreated to take advantage of availability zone support.

Prerequisites

The following are the current requirements/limitations for enabling availability zone support:

- The search service must be in a region that supports availability zones
- The search service must be created after availability zone support was enabled in its region.
- The search service must have at least two replicas

Downtime requirements

Downtime will be dependent on how you decide to carry out the migration. Migration will consist of a side-by-side deployment where you'll create a new search service. Downtime will depend on how you choose to redirect traffic from your old search service to your new availability zone enabled search service. For example, if you're using Azure Front Door, downtime will be dependent on the time it takes to update Azure Front Door with your new search service's information. Alternatively, you can route traffic to multiple search services at the same time using Azure Traffic Manager.

Migration guidance: Recreate your search service

When to recreate your search service

If you created your search service in a region that supports availability zones before this support was enabled, you'll need to recreate the search service.

How to recreate your search service

1. Create a new search service in the same region as the old search service. This region should support availability zones on or after the current date.

   > **Important**
   >
   > The **free and basic tiers do not support availability zones**, and so they should not be used.

2. Add at least two replicas to your new search service. Once the search service has at least two replicas, it automatically takes advantage of availability zone support.

3. Migrate your data from your old search service to your new search service by rebuilding of all your search indexes from your old service.

To rebuild all of your search indexes, choose one of the following two options:

- Move individual indexes from your old search service to your new one
- Rebuild indexes from an external data source if one is available.

1. Redirect traffic from your old search service to your new search service. This may require updates to your application that uses the old search service.

> **Tip**
>
> Services such as **Azure Front Door** and **Azure Traffic Manager** help simplify this process.

Next steps

Learn how to create and deploy ARM templates

ARM Quickstart Templates

Learn about high availability in Azure Cognitive Search

Additional resources

Documentation

Introduction to Azure Cognitive Search - Azure Cognitive Search

Azure Cognitive Search is a fully managed cloud search service from Microsoft. Learn about use cases, the development workflow, comparisons to other Microsoft search products, and how to get started.

Multitenancy and content isolation - Azure Cognitive Search

Learn about common design patterns for multitenant SaaS applications while using Azure Cognitive Search.

Availability and continuity - Azure Cognitive Search

learn how to make a search service highly available and resilient against period disruptions or even catastrophic failures.

Plan and manage costs - Azure Cognitive Search

Learn about billable events, the billing model, and tips for cost control when running a Cognitive Search service.

Security overview - Azure Cognitive Search

Learn about the security features in Azure Cognitive Search to protect endpoints, content, and operations.

Quickstart: Create a knowledge store in the Azure portal - Azure Cognitive Search

Use the Import data wizard to create a knowledge store used for persisting enriched content. Connect to a knowledge store for analysis from other apps, or send enriched content to downstream processes.

Scale and manage custom skill - Azure Cognitive Search

Learn the tools and techniques for efficiently scaling out a custom skill for maximum throughput. Custom skills invoke custom AI models or logic that you can add to an AI-enriched indexing pipeline in Azure Cognitive Search.

Data sources gallery - Azure Cognitive Search

Lists all of the supported data sources for importing into an Azure Cognitive Search index.

Show 5 more

Design patterns for multitenant SaaS applications and Azure Cognitive Search

Article • 09/15/2022

A multitenant application is one that provides the same services and capabilities to any number of tenants who can't see or share the data of any other tenant. This article discusses tenant isolation strategies for multitenant applications built with Azure Cognitive Search.

Azure Cognitive Search concepts

As a search-as-a-service solution, Azure Cognitive Search allows developers to add rich search experiences to applications without managing any infrastructure or becoming an expert in information retrieval. Data is uploaded to the service and then stored in the cloud. Using simple requests to the Azure Cognitive Search API, the data can then be modified and searched.

Search services, indexes, fields, and documents

Before discussing design patterns, it's important to understand a few basic concepts.

When using Azure Cognitive Search, one subscribes to a *search service*. As data is uploaded to Azure Cognitive Search, it's stored in an *index* within the search service. There can be a number of indexes within a single service. To use the familiar concepts of databases, the search service can be likened to a database while the indexes within a service can be likened to tables within a database.

Each index within a search service has its own schema, which is defined by a number of customizable *fields*. Data is added to an Azure Cognitive Search index in the form of individual *documents*. Each document must be uploaded to a particular index and must fit that index's schema. When searching data using Azure Cognitive Search, the full-text search queries are issued against a particular index. To compare these concepts to those of a database, fields can be likened to columns in a table and documents can be likened to rows.

Scalability

Any Azure Cognitive Search service in the Standard pricing tier can scale in two dimensions: storage and availability.

- *Partitions* can be added to increase the storage of a search service.
- *Replicas* can be added to a service to increase the throughput of requests that a search service can handle.

Adding and removing partitions and replicas at will allow the capacity of the search service to grow with the amount of data and traffic the application demands. In order for a search service to achieve a read SLA, it requires two replicas. In order for a service to achieve a read-write SLA, it requires three replicas.

Service and index limits in Azure Cognitive Search

There are a few different pricing tiers in Azure Cognitive Search, each of the tiers has different limits and quotas. Some of these limits are at the service-level, some are at the index-level, and some are at the partition-level.

	Basic	Standard1	Standard2	Standard3	Standard3 HD
Maximum Replicas per Service	3	12	12	12	12
Maximum Partitions per Service	1	12	12	12	3
Maximum Search Units (Replicas*Partitions) per Service	3	36	36	36	36 (max 3 partitions)
Maximum Storage per Service	2 GB	300 GB	1.2 TB	2.4 TB	600 GB
Maximum Storage per Partition	2 GB	25 GB	100 GB	200 GB	200 GB
Maximum Indexes per Service	5	50	200	200	3000 (max 1000 indexes/partition)

S3 High Density

In Azure Cognitive Search's S3 pricing tier, there's an option for the High Density (HD) mode designed specifically for multitenant scenarios. In many cases, it's necessary to support a large number of smaller tenants under a single service to achieve the benefits of simplicity and cost efficiency.

S3 HD allows for the many small indexes to be packed under the management of a single search service by trading the ability to scale out indexes using partitions for the

ability to host more indexes in a single service.

An S3 service is designed to host a fixed number of indexes (maximum 200) and allow each index to scale in size horizontally as new partitions are added to the service. Adding partitions to S3 HD services increases the maximum number of indexes that the service can host. The ideal maximum size for an individual S3HD index is around 50 - 80 GB, although there's no hard size limit on each index imposed by the system.

Considerations for multitenant applications

Multitenant applications must effectively distribute resources among the tenants while preserving some level of privacy between the various tenants. There are a few considerations when designing the architecture for such an application:

- *Tenant isolation:* Application developers need to take appropriate measures to ensure that no tenants have unauthorized or unwanted access to the data of other tenants. Beyond the perspective of data privacy, tenant isolation strategies require effective management of shared resources and protection from noisy neighbors.
- *Cloud resource cost:* As with any other application, software solutions must remain cost competitive as a component of a multitenant application.
- *Ease of Operations:* When developing a multitenant architecture, the impact on the application's operations and complexity is an important consideration. Azure Cognitive Search has a 99.9% SLA.
- *Global footprint:* Multitenant applications may need to effectively serve tenants, which are distributed across the globe.
- *Scalability:* Application developers need to consider how they reconcile between maintaining a sufficiently low level of application complexity and designing the application to scale with number of tenants and the size of tenants' data and workload.

Azure Cognitive Search offers a few boundaries that can be used to isolate tenants' data and workload.

Modeling multitenancy with Azure Cognitive Search

In the case of a multitenant scenario, the application developer consumes one or more search services and divides their tenants among services, indexes, or both. Azure

Cognitive Search has a few common patterns when modeling a multitenant scenario:

- *One index per tenant:* Each tenant has its own index within a search service that is shared with other tenants.
- *One service per tenant:* Each tenant has its own dedicated Azure Cognitive Search service, offering highest level of data and workload separation.
- *Mix of both:* Larger, more-active tenants are assigned dedicated services while smaller tenants are assigned individual indexes within shared services.

Model 1: One index per tenant

In an index-per-tenant model, multiple tenants occupy a single Azure Cognitive Search service where each tenant has their own index.

Tenants achieve data isolation because all search requests and document operations are issued at an index level in Azure Cognitive Search. In the application layer, there's the need awareness to direct the various tenants' traffic to the proper indexes while also managing resources at the service level across all tenants.

A key attribute of the index-per-tenant model is the ability for the application developer to oversubscribe the capacity of a search service among the application's tenants. If the tenants have an uneven distribution of workload, the optimal combination of tenants can be distributed across a search service's indexes to accommodate a number of highly active, resource-intensive tenants while simultaneously serving a long tail of less active tenants. The trade-off is the inability of the model to handle situations where each tenant is concurrently highly active.

The index-per-tenant model provides the basis for a variable cost model, where an entire Azure Cognitive Search service is bought up-front and then subsequently filled

with tenants. This allows for unused capacity to be designated for trials and free accounts.

For applications with a global footprint, the index-per-tenant model may not be the most efficient. If an application's tenants are distributed across the globe, a separate service may be necessary for each region, which may duplicate costs across each of them.

Azure Cognitive Search allows for the scale of both the individual indexes and the total number of indexes to grow. If an appropriate pricing tier is chosen, partitions and replicas can be added to the entire search service when an individual index within the service grows too large in terms of storage or traffic.

If the total number of indexes grows too large for a single service, another service has to be provisioned to accommodate the new tenants. If indexes have to be moved between search services as new services are added, the data from the index has to be manually copied from one index to the other as Azure Cognitive Search doesn't allow for an index to be moved.

Model 2: One service per tenant

In a service-per-tenant architecture, each tenant has its own search service.

In this model, the application achieves the maximum level of isolation for its tenants. Each service has dedicated storage and throughput for handling search requests. Each tenant has individual ownership of API keys.

For applications where each tenant has a large footprint or the workload has little variability from tenant to tenant, the service-per-tenant model is an effective choice as resources aren't shared across various tenants' workloads.

A service per tenant model also offers the benefit of a predictable, fixed cost model. There's no up-front investment in an entire search service until there's a tenant to fill it, however the cost-per-tenant is higher than an index-per-tenant model.

The service-per-tenant model is an efficient choice for applications with a global footprint. With geographically distributed tenants, it's easy to have each tenant's service in the appropriate region.

The challenges in scaling this pattern arise when individual tenants outgrow their service. Azure Cognitive Search doesn't currently support upgrading the pricing tier of a search service, so all data would have to be manually copied to a new service.

Model 3: Hybrid

Another pattern for modeling multitenancy is mixing both index-per-tenant and service-per-tenant strategies.

By mixing the two patterns, an application's largest tenants can occupy dedicated services while the long tail of less active, smaller tenants can occupy indexes in a shared service. This model ensures that the largest tenants have consistently high performance from the service while helping to protect the smaller tenants from any noisy neighbors.

However, implementing this strategy relies on foresight in predicting which tenants will require a dedicated service versus an index in a shared service. Application complexity increases with the need to manage both of these multitenancy models.

Achieving even finer granularity

The above design patterns to model multitenant scenarios in Azure Cognitive Search assume a uniform scope where each tenant is a whole instance of an application. However, applications can sometimes handle many smaller scopes.

If service-per-tenant and index-per-tenant models aren't sufficiently small scopes, it's possible to model an index to achieve an even finer degree of granularity.

To have a single index behave differently for different client endpoints, a field can be added to an index, which designates a certain value for each possible client. Each time a client calls Azure Cognitive Search to query or modify an index, the code from the client application specifies the appropriate value for that field using Azure Cognitive Search's filter capability at query time.

This method can be used to achieve functionality of separate user accounts, separate permission levels, and even completely separate applications.

> **Note**
>
> Using the approach described above to configure a single index to serve multiple tenants affects the relevance of search results. Search relevance scores are computed at an index-level scope, not a tenant-level scope, so all tenants' data is incorporated in the relevance scores' underlying statistics such as term frequency.

Next steps

Azure Cognitive Search is a compelling choice for many applications. When evaluating the various design patterns for multitenant applications, consider the various pricing tiers and the respective service limits to best tailor Azure Cognitive Search to fit application workloads and architectures of all sizes.

Service administration for Azure Cognitive Search in the Azure portal

Article • 01/12/2023 • 4 minutes to read

Azure Cognitive Search is a fully managed, cloud-based search service used for building a rich search experience into custom apps. This article covers the administration tasks that you can perform in the Azure portal for a search service that you've already created.

Depending on your permission level, the portal covers virtually all aspects of search service operations, including:

- Service administration
- Content management
- Content exploration

Each search service is managed as a standalone resource. The following image shows the portal pages for a single free search service called "demo-search-svc".

Overview (home) page

The overview page is the "home" page of each service. In the following screenshot, the areas on the screen enclosed in red boxes indicate tasks, tools, and tiles that you might use often, especially if you're new to the service.

Area	Description
1	The **Essentials** section lists service properties, such as the service endpoint, service tier, and replica and partition counts.
2	A command bar at the top of the page includes Import data and Search explorer, used for prototyping and exploration.
3	Tabbed pages in the center provide quick access to usage statistics, service health metrics, and access to all of the existing indexes, indexers, data sources, and skillsets.
4	Navigation links to other pages.

Read-only service properties

Several aspects of a search service are determined when the service is provisioned and can't be easily changed:

- Service name
- Service location [1]
- Service tier [2]

[1] Although there are ARM and bicep templates for service deployment, moving content is a manual job.

[2] Switching a tier requires creating a new service or filing a support ticket to request a tier upgrade.

Management tasks

Service administration includes the following tasks:

- Adjust capacity by adding or removing replicas and partitions
- Manage API keys used for content access
- Manage Azure roles used for content and service access
- Configure IP firewall rules to restrict access by IP address
- Configure a private endpoint using Azure Private Link and a private virtual network
- Monitor service health and operations: storage, query volumes, and latency

There's feature parity across all modalities and languages except for preview management features. In general, preview management features are released through the Management REST API first. Programmatic support for service administration can be found in the following APIs and modules:

- Management REST API reference
- Az.Search PowerShell module
- az search Azure CLI module

You can also use the management client libraries in the Azure SDKs for .NET, Python, Java, and JavaScript.

Data collection and retention

Because Azure Cognitive Search is a monitored resource, you can review the built-in **activity logs** and **platform metrics** for insights into service operations. Activity logs and the data used to report on platform metrics are retained for the periods described in the following table.

If you opt in for **resource logging**, you'll specify durable storage over which you'll have full control over data retention and data access through Kusto queries. For more information on how to set up resource logging in Cognitive Search, see Collect and analyze log data.

Internally, Microsoft collects telemetry data about your service and the platform. It's stored internally in Microsoft data centers and made globally available to Microsoft support engineers when you open a support ticket.

Monitoring data	Retention
Activity logs	90 days on a rolling schedule
Platform metrics	93 days on a rolling schedule, except that portal visualization is limited to a 30 day window
Resource logs	User-managed
Telemetry	One and a half years

> **Note**
>
> See the **"Data residency"** section of the security overview article for more information about data location and privacy.

Administrator permissions

When you open the search service overview page, the Azure role assigned to your account determines what portal content is available to you. The overview page at the beginning of the article shows the portal content available to an Owner or Contributor.

Azure roles used for service administration include:

- Owner
- Contributor (same as Owner, minus the ability to assign roles)
- Reader (provides access to service information in the Essentials section and in the Monitoring tab)

By default, all search services start with at least one Owner. Owners, service administrators, and co-administrators have permission to create other administrators and other role assignments.

Also by default, search services start with API keys for content-related tasks that an Owner or Contributor might perform in the portal. However, it's possible to turn off API key authentication and use Azure role-based access control exclusively. If you turn off API keys, be sure to set up data access role assignments so that all features in the portal remain operational.

> **Tip**
>
> By default, any owner or administrator can create or delete services. To prevent accidental deletions, you can **lock resources**.

Next steps

- Review monitoring capabilities available in the portal
- Automate with PowerShell or Azure CLI
- Review security features to protect content and operations
- Enable resource logging to monitor query and indexing workloads

Manage your Azure Cognitive Search service with PowerShell

Article • 01/25/2023 • 11 minutes to read

You can run PowerShell cmdlets and scripts on Windows, Linux, or in Azure Cloud Shell to create and configure Azure Cognitive Search. The **Az.Search** module extends Azure PowerShell with full parity to the Search Management REST APIs and the ability to perform the following tasks:

- ✓ List search services in a subscription
- ✓ Return service information
- ✓ Create or delete a service
- ✓ Create a service with a private endpoint
- ✓ Regenerate admin API-keys
- ✓ Create or delete query api-keys
- ✓ Scale up or down with replicas and partitions
- ✓ Create a shared private link resource

Occasionally, questions are asked about tasks *not* on the above list.

You cannot change a server name, region, or tier programmatically or in the portal. Dedicated resources are allocated when a service is created. As such, changing the underlying hardware (location or node type) requires a new service.

You cannot use tools or APIs to transfer content, such as an index, from one service to another. Within a service, programmatic creation of content is through Search Service REST API or an SDK such as Azure SDK for .NET. While there are no dedicated commands for content migration, you can write script that calls REST API or a client library to create and load indexes on a new service.

Preview administration features are typically not available in the **Az.Search** module. If you want to use a preview feature, use the Management REST API and a preview API version.

Check versions and load modules

The examples in this article are interactive and require elevated permissions. Local PowerShell and the Azure PowerShell (the **Az** module) are required.

PowerShell version check

PowerShell 7.0.6 LTS, PowerShell 7.1.3, or higher is the recommended version of PowerShell for use with the Azure Az PowerShell module on all platforms. Install the latest version of PowerShell if you don't have it.

Azure PowerShell

```
$PSVersionTable.PSVersion
```

Load Azure PowerShell

If you aren't sure whether **Az** is installed, run the following command as a verification step.

Azure PowerShell

```
Get-InstalledModule -Name Az
```

Some systems do not auto-load modules. If you get an error on the previous command, try loading the module, and if that fails, go back to the installation Azure PowerShell installation instructions to see if you missed a step.

Azure PowerShell

```
Import-Module -Name Az
```

Connect to Azure with a browser sign-in token

You can use your portal sign-in credentials to connect to a subscription in PowerShell. Alternatively you can authenticate non-interactively with a service principal.

Azure PowerShell

```
Connect-AzAccount
```

If you hold multiple Azure subscriptions, set your Azure subscription. To see a list of your current subscriptions, run this command.

Azure PowerShell

```
Get-AzSubscription | sort SubscriptionName | Select SubscriptionName
```

To specify the subscription, run the following command. In the following example, the subscription name is `ContosoSubscription`.

Azure PowerShell

```
Select-AzSubscription -SubscriptionName ContosoSubscription
```

List services in a subscription

The following commands are from **Az.Resources**, returning information about existing resources and services already provisioned in your subscription. If you don't know how many search services are already created, these commands return that information, saving you a trip to the portal.

The first command returns all search services.

Azure PowerShell

```
Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
```

From the list of services, return information about a specific resource.

Azure PowerShell

```
Get-AzResource -ResourceName <service-name>
```

Results should look similar to the following output.

```
Name              : my-demo-searchapp
ResourceGroupName : demo-westus
ResourceType      : Microsoft.Search/searchServices
Location          : westus
ResourceId        : /subscriptions/<alphanumeric-subscription-
ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-
demo-searchapp
```

Import Az.Search

Commands from **Az.Search** are not available until you load the module.

Azure PowerShell

```
Install-Module -Name Az.Search
```

List all Az.Search commands

As a verification step, return a list of commands provided in the module.

Azure PowerShell

```
Get-Command -Module Az.Search
```

Results should look similar to the following output.

```
CommandType     Name                                               Version
Source
-----------     ----                                               -------
------
Cmdlet          Get-AzSearchAdminKeyPair                           0.8.0
Az.Search
Cmdlet          Get-AzSearchPrivateEndpointConnection              0.8.0
Az.Search
Cmdlet          Get-AzSearchPrivateLinkResource                    0.8.0
Az.Search
Cmdlet          Get-AzSearchQueryKey                               0.8.0
Az.Search
Cmdlet          Get-AzSearchService                                0.8.0
Az.Search
Cmdlet          Get-AzSearchSharedPrivateLinkResource              0.8.0
Az.Search
Cmdlet          New-AzSearchAdminKey                               0.8.0
Az.Search
Cmdlet          New-AzSearchQueryKey                               0.8.0
Az.Search
Cmdlet          New-AzSearchService                                0.8.0
Az.Search
Cmdlet          New-AzSearchSharedPrivateLinkResource              0.8.0
Az.Search
Cmdlet          Remove-AzSearchPrivateEndpointConnection           0.8.0
Az.Search
Cmdlet          Remove-AzSearchQueryKey                            0.8.0
Az.Search
Cmdlet          Remove-AzSearchService                             0.8.0
Az.Search
Cmdlet          Remove-AzSearchSharedPrivateLinkResource           0.8.0
Az.Search
Cmdlet          Set-AzSearchPrivateEndpointConnection              0.8.0
Az.Search
Cmdlet          Set-AzSearchService                                0.8.0
Az.Search
```

```
Cmdlet          Set-AzSearchSharedPrivateLinkResource              0.8.0
Az.Search
```

If you have an older version of the package, update the module to get the latest functionality.

Azure PowerShell

```
Update-Module -Name Az.Search
```

Get search service information

After **Az.Search** is imported and you know the resource group containing your search service, run Get-AzSearchService to return the service definition, including name, region, tier, and replica and partition counts. For this command, provide the resource group that contains the search service.

Azure PowerShell

```
Get-AzSearchService -ResourceGroupName <resource-group-name>
```

Results should look similar to the following output.

```
Name              : my-demo-searchapp
ResourceGroupName : demo-westus
ResourceType      : Microsoft.Search/searchServices
Location          : West US
Sku               : Standard
ReplicaCount      : 1
PartitionCount    : 1
HostingMode       : Default
ResourceId        : /subscriptions/<alphanumeric-subscription-
ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-
demo-searchapp
```

Create or delete a service

New-AzSearchService is used to create a new search service.

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> -Name <search-service-name> -Sku "Standard" -Location "West US" -PartitionCount 3 -ReplicaCount 3 -HostingMode Default
```

Results should look similar to the following output.

Azure PowerShell

```
ResourceGroupName : demo-westus
Name              : my-demo-searchapp
Id                : /subscriptions/<alphanumeric-subscription-ID>/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
Location          : West US
Sku               : Standard
ReplicaCount      : 3
PartitionCount    : 3
HostingMode       : Default
Tags
```

Remove-AzSearchService is used to delete a service and its data.

Azure PowerShell

```
Remove-AzSearchService -ResourceGroupName <resource-group-name> -Name <search-service-name>
```

You'll be asked to confirm the action.

Azure PowerShell

```
Confirm
Are you sure you want to remove Search Service 'pstestazuresearch01'?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"): y
```

Create a service with IP rules

Depending on your security requirements, you may want to create a search service with an IP firewall configured. To do so, first define the IP Rules and then pass them to the `IPRuleList` parameter as shown below.

Azure PowerShell

```
$ipRules = @([pscustomobject]@{Value="55.5.63.73"},
        [pscustomobject]@{Value="52.228.215.197"},
        [pscustomobject]@{Value="101.37.221.205"})
```

```
New-AzSearchService -ResourceGroupName <resource-group-name> `
                    -Name <search-service-name> `
                    -Sku Standard `
                    -Location "West US" `
                    -PartitionCount 3 -ReplicaCount 3 `
                    -HostingMode Default `
                    -IPRuleList $ipRules
```

Create a service with a system assigned managed identity

In some cases, such as when using managed identity to connect to a data source, you will need to turn on system assigned managed identity. This is done by adding `-IdentityType SystemAssigned` to the command.

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> `
                    -Name <search-service-name> `
                    -Sku Standard `
                    -Location "West US" `
                    -PartitionCount 3 -ReplicaCount 3 `
                    -HostingMode Default `
                    -IdentityType SystemAssigned
```

Create an S3HD service

To create an S3HD service, a combination of `-Sku` and `-HostingMode` is used. Set `-Sku` to `Standard3` and `-HostingMode` to `HighDensity`.

Azure PowerShell

```
New-AzSearchService -ResourceGroupName <resource-group-name> `
                    -Name <search-service-name> `
                    -Sku Standard3 `
                    -Location "West US" `
                    -PartitionCount 1 -ReplicaCount 3 `
                    -HostingMode HighDensity
```

Create a service with a private endpoint

Private Endpoints for Azure Cognitive Search allow a client on a virtual network to securely access data in a search index over a Private Link. The private endpoint uses an IP address from the virtual network address space for your search service. Network traffic between the client and the search service traverses over the virtual network and a

private link on the Microsoft backbone network, eliminating exposure from the public internet. For more details, see Creating a private endpoint for Azure Cognitive Search

The following example shows how to create a search service with a private endpoint.

First, deploy a search service with `PublicNetworkAccess` set to `Disabled`.

Azure PowerShell

```
$searchService = New-AzSearchService `
    -ResourceGroupName <search-service-resource-group-name> `
    -Name <search-service-name> `
    -Sku Standard `
    -Location "West US" `
    -PartitionCount 1 -ReplicaCount 1 `
    -HostingMode Default `
    -PublicNetworkAccess Disabled
```

Next, create a virtual network, private network connection, and the private endpoint.

Azure PowerShell

```
# Create the subnet
$subnetConfig = New-AzVirtualNetworkSubnetConfig `
    -Name <subnet-name> `
    -AddressPrefix 10.1.0.0/24 `
    -PrivateEndpointNetworkPolicies Disabled

# Create the virtual network
$virtualNetwork = New-AzVirtualNetwork `
    -ResourceGroupName <vm-resource-group-name> `
    -Location "West US" `
    -Name <virtual-network-name> `
    -AddressPrefix 10.1.0.0/16 `
    -Subnet $subnetConfig

# Create the private network connection
$privateLinkConnection = New-AzPrivateLinkServiceConnection `
    -Name <private-link-name> `
    -PrivateLinkServiceId $searchService.Id `
    -GroupId searchService

# Create the private endpoint
$privateEndpoint = New-AzPrivateEndpoint `
    -Name <private-endpoint-name> `
    -ResourceGroupName <private-endpoint-resource-group-name> `
    -Location "West US" `
    -Subnet $virtualNetwork.subnets[0] `
    -PrivateLinkServiceConnection $privateLinkConnection
```

Finally, create a private DNS Zone.

Azure PowerShell

```
## Create private dns zone
$zone = New-AzPrivateDnsZone `
    -ResourceGroupName <private-dns-resource-group-name> `
    -Name "privatelink.search.windows.net"

## Create dns network link
$link = New-AzPrivateDnsVirtualNetworkLink `
    -ResourceGroupName <private-dns-link-resource-group-name> `
    -ZoneName "privatelink.search.windows.net" `
    -Name "myLink" `
    -VirtualNetworkId $virtualNetwork.Id

## Create DNS configuration
$config = New-AzPrivateDnsZoneConfig `
    -Name "privatelink.search.windows.net" `
    -PrivateDnsZoneId $zone.ResourceId

## Create DNS zone group
New-AzPrivateDnsZoneGroup `
    -ResourceGroupName <private-dns-zone-resource-group-name> `
    -PrivateEndpointName <private-endpoint-name> `
    -Name 'myZoneGroup' `
    -PrivateDnsZoneConfig $config
```

For more details on creating private endpoints in PowerShell, see this Private Link Quickstart

Manage private endpoint connections

In addition to creating a private endpoint connection, you can also `Get`, `Set`, and `Remove` the connection.

Get-AzSearchPrivateEndpointConnection is used to retrieve a private endpoint connection and to see its status.

Azure PowerShell

```
Get-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name>
```

Set-AzSearchPrivateEndpointConnection is used to update the connection. The following example sets a private endpoint connection to rejected:

Azure PowerShell

```
Set-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <pe-connection-name> -Status Rejected  -Description "Rejected"
```

Remove-AzSearchPrivateEndpointConnection is used to delete the private endpoint connection.

Azure PowerShell

```
 Remove-AzSearchPrivateEndpointConnection -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -Name <pe-connection-name>
```

Regenerate admin keys

New-AzSearchAdminKey is used to roll over admin API keys. Two admin keys are created with each service for authenticated access. Keys are required on every request. Both admin keys are functionally equivalent, granting full write access to a search service with the ability to retrieve any information, or create and delete any object. Two keys exist so that you can use one while replacing the other.

You can only regenerate one at a time, specified as either the `primary` or `secondary` key. For uninterrupted service, remember to update all client code to use a secondary key while rolling over the primary key. Avoid changing the keys while operations are in flight.

As you might expect, if you regenerate keys without updating client code, requests using the old key will fail. Regenerating all new keys does not permanently lock you out of your service, and you can still access the service through the portal. After you regenerate primary and secondary keys, you can update client code to use the new keys and operations will resume accordingly.

Values for the API keys are generated by the service. You cannot provide a custom key for Azure Cognitive Search to use. Similarly, there is no user-defined name for admin API-keys. References to the key are fixed strings, either `primary` or `secondary`.

Azure PowerShell

```
New-AzSearchAdminKey -ResourceGroupName <search-service-resource-group-name> -ServiceName <search-service-name> -KeyKind Primary
```

Results should look similar to the following output. Both keys are returned even though you only change one at a time.

```
Primary                     Secondary
-------                     ---------
<alphanumeric-guid>         <alphanumeric-guid>
```

Create or delete query keys

New-AzSearchQueryKey is used to create query API keys for read-only access from client apps to an Azure Cognitive Search index. Query keys are used to authenticate to a specific index for the purpose of retrieving search results. Query keys do not grant read-only access to other items on the service, such as an index, data source, or indexer.

You cannot provide a key for Azure Cognitive Search to use. API keys are generated by the service.

Azure PowerShell

```
New-AzSearchQueryKey -ResourceGroupName <search-service-resource-group-name>
-ServiceName <search-service-name> -Name <query-key-name>
```

Scale replicas and partitions

Set-AzSearchService is used to increase or decrease replicas and partitions to readjust billable resources within your service. Increasing replicas or partitions adds to your bill, which has both fixed and variable charges. If you have a temporary need for additional processing power, you can increase replicas and partitions to handle the workload. The monitoring area in the Overview portal page has tiles on query latency, queries per second, and throttling, indicating whether current capacity is adequate.

It can take a while to add or remove resourcing. Adjustments to capacity occur in the background, allowing existing workloads to continue. Additional capacity is used for incoming requests as soon as it's ready, with no additional configuration required.

Removing capacity can be disruptive. Stopping all indexing and indexer jobs prior to reducing capacity is recommended to avoid dropped requests. If that isn't feasible, you might consider reducing capacity incrementally, one replica and partition at a time, until your new target levels are reached.

Once you submit the command, there is no way to terminate it midway through. You will have to wait until the command is finished before revising the counts.

Azure PowerShell

```
Set-AzSearchService -ResourceGroupName <search-service-resource-group-name>
-Name <search-service-name> -PartitionCount 6 -ReplicaCount 6
```

Results should look similar to the following output.

```
ResourceGroupName : demo-westus
Name              : my-demo-searchapp
Location          : West US
Sku               : Standard
ReplicaCount      : 6
PartitionCount    : 6
HostingMode       : Default
Id                : /subscriptions/65a1016d-0f67-45d2-b838-
b8f373d6d52e/resourceGroups/demo-
westus/providers/Microsoft.Search/searchServices/my-demo-searchapp
```

Create a shared private link resource

Private endpoints of secured resources that are created through Azure Cognitive Search APIs are referred to as *shared private link resources*. This is because you're "sharing" access to a resource, such as a storage account, that has been integrated with the Azure Private Link service.

If you're using an indexer to index data in Azure Cognitive Search, and your data source is on a private network, you can create an outbound private endpoint connection to reach the data.

A full list of the Azure Resources for which you can create outbound private endpoints from Azure Cognitive Search can be found here along with the related **Group ID** values.

New-AzSearchSharedPrivateLinkResource is used to create the shared private link resource. Keep in mind that some configuration may be required for the data source before running this command.

Azure PowerShell

```
New-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-
serviceresource-group-name> -ServiceName <search-service-name> -Name <spl-
name> -PrivateLinkResourceId /subscriptions/<alphanumeric-subscription-
```

```
ID>/resourceGroups/<storage-resource-group-
name>/providers/Microsoft.Storage/storageAccounts/myBlobStorage -GroupId
<group-id> -RequestMessage "Please approve"
```

Get-AzSearchSharedPrivateLinkResource allows you to retrieve the shared private link resources and view their status.

Azure PowerShell

```
Get-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-service-
resource-group-name> -ServiceName <search-service-name> -Name <spl-name>
```

You'll need to approve the connection with the following command before it can be used.

Azure PowerShell

```
Approve-AzPrivateEndpointConnection `
    -Name <spl-name> `
    -ServiceName <search-service-name> `
    -ResourceGroupName <search-service-resource-group-name> `
    -Description = "Approved"
```

Remove-AzSearchSharedPrivateLinkResource is used to delete the shared private link resource.

Azure PowerShell

```
$job = Remove-AzSearchSharedPrivateLinkResource -ResourceGroupName <search-
service-resource-group-name> -ServiceName <search-service-name> -Name <spl-
name> -Force -AsJob

$job | Get-Job
```

For full details on setting up shared private link resources, see the documentation on making indexer connections through a private endpoint.

Next steps

Build an index, query an index using the portal, REST APIs, or the .NET SDK.

- Create an Azure Cognitive Search index in the Azure portal
- Set up an indexer to load data from other services
- Query an Azure Cognitive Search index using Search explorer in the Azure portal
- How to use Azure Cognitive Search in .NET

Additional resources

Documentation

Azure CLI scripts using the az search module - Azure Cognitive Search

Create and configure an Azure Cognitive Search service with the Azure CLI. You can scale a service up or down, manage admin and query api-keys, and query for system information.

List Indexes (Azure Cognitive Search REST API)

Return a list of index names for indexes in the current Azure Cognitive Search service.

az search

az search service

Quickstart: Create a search index in PowerShell using REST APIs - Azure Cognitive Search

In this REST API quickstart, learn how to create an index, load data, and run queries using PowerShell's Invoke-RestMethod and the Azure Cognitive Search REST API.

REST APIs for search management - Azure Cognitive Search

Create and configure an Azure Cognitive Search service with the Management REST API. The Management REST API is comprehensive in scope, with access to generally available and preview features.

Connect through firewalls - Azure Cognitive Search

Configure IP firewall rules to allow data access by an Azure Cognitive Search indexer.

Quickstart: Deploy using templates - Azure Cognitive Search

You can quickly deploy an Azure Cognitive Search service instance using the Azure Resource Manager template.

Show 5 more

Training

Learning paths and modules

Design and implement private access to Azure Services - Training

Design and implement private access to Azure Services

Manage your Azure Cognitive Search service with the Azure CLI

Article • 03/10/2023 • 11 minutes to read

You can run Azure CLI commands and scripts on Windows, macOS, Linux, or in Azure Cloud Shell to create and configure Azure Cognitive Search. The **az search** module extends the Azure CLI with full parity to the Search Management REST APIs and the ability to perform the following tasks:

- ✓ List search services in a subscription
- ✓ Return service information
- ✓ Create or delete a service
- ✓ Create a service with a private endpoint
- ✓ Regenerate admin API-keys
- ✓ Create or delete query api-keys
- ✓ Scale up or down with replicas and partitions
- ✓ Create a shared private link resource

Occasionally, questions are asked about tasks *not* on the above list.

You cannot change a server name, region, or tier programmatically or in the portal. Dedicated resources are allocated when a service is created. As such, changing the underlying hardware (location or node type) requires a new service.

You cannot use tools or APIs to transfer content, such as an index, from one service to another. Within a service, programmatic creation of content is through Search Service REST API or an SDK such as Azure SDK for .NET. While there are no dedicated commands for content migration, you can write script that calls REST API or a client library to create and load indexes on a new service.

Preview administration features are typically not available in the **az search** module. If you want to use a preview feature, use the Management REST API and a preview API version.

Prerequisites

- Use the Bash environment in Azure Cloud Shell. For more information, see Quickstart for Bash in Azure Cloud Shell.

  Launch Cloud Shell

- If you prefer to run CLI reference commands locally, install the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see How to run the Azure CLI in a Docker container.
    - If you're using a local installation, sign in to the Azure CLI by using the az login command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see Sign in with the Azure CLI.
    - When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see Use extensions with the Azure CLI.
    - Run az version to find the version and dependent libraries that are installed. To upgrade to the latest version, run az upgrade.

List services in a subscription

The following commands are from **az resource**, returning information about existing resources and services already provisioned in your subscription. If you don't know how many search services are already created, these commands return that information, saving you a trip to the portal.

The first command returns all search services.

Azure CLI

```
az resource list --resource-type Microsoft.Search/searchServices --output
table
```

From the list of services, return information about a specific resource.

Azure CLI

```
az resource list --name <search-service-name>
```

List all az search commands

You can view information on the subgroups and commands available in **az search** from within the CLI. Alternatively, you can review the documentation.

To view the subgroups available within `az search`, run the following command.

Azure CLI

```
az search --help
```

The response should look similar to the following output.

```
Group
    az search : Manage Azure Search services, admin keys and query keys.
        WARNING: This command group is in preview and under development.
Reference and support
        levels: https://aka.ms/CLI_refstatus
Subgroups:
    admin-key                    : Manage Azure Search admin keys.
    private-endpoint-connection  : Manage Azure Search private endpoint
connections.
    private-link-resource        : Manage Azure Search private link
resources.
    query-key                    : Manage Azure Search query keys.
    service                      : Manage Azure Search services.
    shared-private-link-resource : Manage Azure Search shared private link
resources.

For more specific examples, use: az find "az search"
```

Within each subgroup, multiple commands are available. You can see the available commands for the `service` subgroup by running the following line.

Azure CLI

```
az search service --help
```

You can also see the arguments available for a particular command.

Azure CLI

```
az search service create --help
```

Get search service information

If you know the resource group containing your search service, run **az search service show** to return the service definition, including name, region, tier, and replica and partition counts. For this command, provide the resource group that contains the search service.

Azure CLI

```
az search service show --name <service-name> --resource-group <search-service-resource-group-name>
```

Create or delete a service

To create a new search service, use the **az search service create** command.

Azure CLI

```
az search service create \
    --name <service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1
```

Results should look similar to the following output:

```
{
  "hostingMode": "default",
  "id": "/subscriptions/<alphanumeric-subscription-ID>/resourceGroups/demo-westus/providers/Microsoft.Search/searchServices/my-demo-searchapp",
  "identity": null,
  "location": "West US",
  "name": "my-demo-searchapp",
  "networkRuleSet": {
    "bypass": "None",
    "ipRules": []
  },
  "partitionCount": 1,
  "privateEndpointConnections": [],
  "provisioningState": "succeeded",
  "publicNetworkAccess": "Enabled",
  "replicaCount": 1,
  "resourceGroup": "demo-westus",
  "sharedPrivateLinkResources": [],
  "sku": {
    "name": "standard"
  },
  "status": "running",
  "statusDetails": "",
  "tags": null,
  "type": "Microsoft.Search/searchServices"
}
```

az search service delete removes the service and its data.

Azure CLI

```
az search service delete --name <service-name> \
                         --resource-group  <search-service-resource-group-name> \
```

Create a service with IP rules

Depending on your security requirements, you may want to create a search service with an IP firewall configured. To do so, pass the Public IP (v4) addresses or CIDR ranges to the `ip-rules` argument as shown below. Rules should be separated by a comma (`,`) or semicolon (`;`).

Azure CLI

```
az search service create \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1 \
    --ip-rules "55.5.63.73;52.228.215.197;101.37.221.205"
```

Create a service with a system assigned managed identity

In some cases, such as when using managed identity to connect to a data source, you will need to turn on system assigned managed identity. This is done by adding `--identity-type SystemAssigned` to the command.

Azure CLI

```
az search service create \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1 \
    --identity-type SystemAssigned
```

Create a service with a private endpoint

Private Endpoints for Azure Cognitive Search allow a client on a virtual network to securely access data in a search index over a Private Link. The private endpoint uses an IP address from the virtual network address space for your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. For more details, please refer to the documentation on creating a private endpoint for Azure Cognitive Search

The following example shows how to create a search service with a private endpoint.

First, deploy a search service with `PublicNetworkAccess` set to `Disabled`.

Azure CLI

```
az search service create \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --sku Standard \
    --partition-count 1 \
    --replica-count 1 \
    --public-access Disabled
```

Next, create a virtual network and the private endpoint.

Azure CLI

```
# Create the virtual network
az network vnet create \
    --resource-group <vnet-resource-group-name> \
    --location "West US" \
    --name <virtual-network-name> \
    --address-prefixes 10.1.0.0/16 \
    --subnet-name <subnet-name> \
    --subnet-prefixes 10.1.0.0/24

# Update the subnet to disable private endpoint network policies
az network vnet subnet update \
    --name <subnet-name> \
    --resource-group <vnet-resource-group-name> \
    --vnet-name <virtual-network-name> \
    --disable-private-endpoint-network-policies true

# Get the id of the search service
id=$(az search service show \
    --resource-group <search-service-resource-group-name> \
    --name <search-service-name> \
    --query [id] \
    --output tsv)

# Create the private endpoint
```

```
az network private-endpoint create \
    --name <private-endpoint-name> \
    --resource-group <private-endpoint-resource-group-name> \
    --vnet-name <virtual-network-name> \
    --subnet <subnet-name> \
    --private-connection-resource-id $id \
    --group-id searchService \
    --connection-name <private-link-connection-name>
```

Finally, create a private DNS Zone.

Azure CLI

```
## Create private DNS zone
az network private-dns zone create \
    --resource-group <private-dns-resource-group-name> \
    --name "privatelink.search.windows.net"

## Create DNS network link
az network private-dns link vnet create \
    --resource-group <private-dns-resource-group-name> \
    --zone-name "privatelink.search.windows.net" \
    --name "myLink" \
    --virtual-network <virtual-network-name> \
    --registration-enabled false

## Create DNS zone group
az network private-endpoint dns-zone-group create \
   --resource-group <private-endpoint-resource-group-name>\
   --endpoint-name <private-endpoint-name> \
   --name "myZoneGroup" \
   --private-dns-zone "privatelink.search.windows.net" \
   --zone-name "searchServiceZone"
```

For more information on creating private endpoints in Azure CLI, see this Private Link Quickstart

Manage private endpoint connections

In addition to creating a private endpoint connection, you can also `show`, `update`, and `delete` the connection.

To retrieve a private endpoint connection and to see its status, use **az search private-endpoint-connection show**.

Azure CLI

```
az search private-endpoint-connection show \
    --name <pe-connection-name> \
```

```
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name>
```

To update the connection, use **az search private-endpoint-connection update**. The following example sets a private endpoint connection to rejected:

Azure CLI

```
az search private-endpoint-connection show \
    --name <pe-connection-name> \
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name>
    --status Rejected \
    --description "Rejected" \
    --actions-required "Please fix XYZ"
```

To delete the private endpoint connection, use **az search private-endpoint-connection delete**.

Azure CLI

```
az search private-endpoint-connection delete \
    --name <pe-connection-name> \
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name>
```

Regenerate admin keys

To roll over admin API keys, use **az search admin-key renew**. Two admin keys are created with each service for authenticated access. Keys are required on every request. Both admin keys are functionally equivalent, granting full write access to a search service with the ability to retrieve any information, or create and delete any object. Two keys exist so that you can use one while replacing the other.

You can only regenerate one at a time, specified as either the `primary` or `secondary` key. For uninterrupted service, remember to update all client code to use a secondary key while rolling over the primary key. Avoid changing the keys while operations are in flight.

As you might expect, if you regenerate keys without updating client code, requests using the old key will fail. Regenerating all new keys does not permanently lock you out of your service, and you can still access the service through the portal. After you regenerate primary and secondary keys, you can update client code to use the new keys and operations will resume accordingly.

Values for the API keys are generated by the service. You cannot provide a custom key for Azure Cognitive Search to use. Similarly, there is no user-defined name for admin API-keys. References to the key are fixed strings, either `primary` or `secondary`.

Azure CLI

```
az search admin-key renew \
    --resource-group <search-service-resource-group-name> \
    --service-name <search-service-name> \
    --key-kind primary
```

Results should look similar to the following output. Both keys are returned even though you only change one at a time.

```
{
  "primaryKey": <alphanumeric-guid>,
  "secondaryKey": <alphanumeric-guid>
}
```

Create or delete query keys

To create query API keys for read-only access from client apps to an Azure Cognitive Search index, use **az search query-key create**. Query keys are used to authenticate to a specific index for the purpose of retrieving search results. Query keys do not grant read-only access to other items on the service, such as an index, data source, or indexer.

You cannot provide a key for Azure Cognitive Search to use. API keys are generated by the service.

Azure CLI

```
az search query-key create \
    --name myQueryKey \
    --resource-group <search-service-resource-group-name> \
    --service-name <search-service-name>
```

Scale replicas and partitions

To increase or decrease replicas and partitions use **az search service update**. Increasing replicas or partitions adds to your bill, which has both fixed and variable charges. If you have a temporary need for additional processing power, you can increase replicas and

partitions to handle the workload. The monitoring area in the Overview portal page has tiles on query latency, queries per second, and throttling, indicating whether current capacity is adequate.

It can take a while to add or remove resourcing. Adjustments to capacity occur in the background, allowing existing workloads to continue. Additional capacity is used for incoming requests as soon as it's ready, with no additional configuration required.

Removing capacity can be disruptive. Stopping all indexing and indexer jobs prior to reducing capacity is recommended to avoid dropped requests. If that isn't feasible, you might consider reducing capacity incrementally, one replica and partition at a time, until your new target levels are reached.

Once you submit the command, there is no way to terminate it midway through. You will have to wait until the command is finished before revising the counts.

Azure CLI

```
az search service update \
    --name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --partition-count 6 \
    --replica-count 6
```

In addition to updating replica and partition counts, you can also update `ip-rules`, `public-access`, and `identity-type`.

Create a shared private link resource

Private endpoints of secured resources that are created through Azure Cognitive Search APIs are referred to as *shared private link resources*. This is because you're "sharing" access to a resource, such as a storage account, that has been integrated with the Azure Private Link service.

If you're using an indexer to index data in Azure Cognitive Search, and your data source is on a private network, you can create an outbound private endpoint connection to reach the data.

A full list of the Azure Resources for which you can create outbound private endpoints from Azure Cognitive Search can be found here along with the related **Group ID** values.

To create the shared private link resource, use **az search shared-private-link-resource create**. Keep in mind that some configuration may be required for the data source before running this command.

Azure CLI

```
az search shared-private-link-resource create \
    --name <spl-name> \
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name> \
    --group-id blob \
    --resource-id "/subscriptions/<alphanumeric-subscription-
ID>/resourceGroups/<resource-group-
name>/providers/Microsoft.Storage/storageAccounts/myBlobStorage"  \
    --request-message "Please approve"
```

To retrieve the shared private link resources and view their status, use **az search shared-private-link-resource list**.

Azure CLI

```
az search shared-private-link-resource list \
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name>
```

You'll need to approve the connection with the following command before it can be used. The ID of the private endpoint connection will need to be retrieved from the child resource. In this case, we get the connection ID from az storage.

Azure CLI

```
id = (az storage account show -n myBlobStorage --query
"privateEndpointConnections[0].id")

az network private-endpoint-connection approve --id $id
```

To delete the shared private link resource, use **az search shared-private-link-resource delete**.

Azure CLI

```
az search shared-private-link-resource delete \
    --name <spl-name> \
    --service-name <search-service-name> \
    --resource-group <search-service-resource-group-name>
```

For full details on setting up shared private link resources, see the documentation on making indexer connections through a private endpoint.

Next steps

Build an index, query an index using the portal, REST APIs, or the .NET SDK.

- Create an Azure Cognitive Search index in the Azure portal
- Set up an indexer to load data from other services
- Query an Azure Cognitive Search index using Search explorer in the Azure portal
- How to use Azure Cognitive Search in .NET

Additional resources

Documentation

az search

Manage with REST - Azure Cognitive Search

Create and configure an Azure Cognitive Search service with the Management REST API. The Management REST API is comprehensive in scope, with access to generally available and preview features.

List Indexes (Azure Cognitive Search REST API)

Return a list of index names for indexes in the current Azure Cognitive Search service.

PowerShell scripts using Az.Search module - Azure Cognitive Search

Create and configure an Azure Cognitive Search service with PowerShell. You can scale a service up or down, manage admin and query api-keys, and query for system information.

az search service

API versions of Search REST APIs - Azure Cognitive Search

Lists the generally available and preview versions of the Search REST APIs for Azure Cognitive Search.

Connect using a managed identity - Azure Cognitive Search

Create a managed identity for your search service and use Azure Active Directory authentication and role-based-access controls for connections to other cloud services.

Get Indexer Status (Azure Cognitive Search REST API)

Return current and historical status notifications generated by an Azure Cognitive Search indexer when it crawls an external data source for searchable content.

Show 5 more

Training

Certification

Microsoft Certified: Azure Administrator Associate - Certifications

Azure administrators implement, manage, and monitor an organization's Microsoft Azure environment, including virtual networks, storage, compute, identity, security, and governance.

Manage your Azure Cognitive Search service with REST APIs

Article • 05/14/2023

In this article, learn how to create and configure an Azure Cognitive Search service using the Management REST APIs. Only the Management REST APIs are guaranteed to provide early access to preview features.

The Management RESt API is available in stable and preview versions. Be sure to set a preview API version if you're accessing preview features.

- ✓ List search services
- ✓ Create or update a service
- ✓ Enable Azure role-based access control for data plane
- ✓ (preview) Enforce a customer-managed key policy
- ✓ (preview) Disable semantic search
- ✓ (preview) Disable workloads that push data to external resources

All of the Management REST APIs have examples. If a task isn't covered in this article, see the API reference instead.

Prerequisites

- An Azure subscription - Create one for free.
- Postman or another REST client that sends HTTP requests.
- Azure CLI used to set up a security principle for the client. You must have owner or administrator permissions to create a security principle.

Create a security principal

Management REST API calls are authenticated through Azure Active Directory (Azure AD). You'll need a security principal for your REST client, along with permissions to create and configure a resource. This section explains how to create a security principal and assign a role.

> ⓘ **Note**

The following steps are borrowed from the **Azure REST APIs with Postman** blog post.

1. Open a command shell for Azure CLI. If you don't have Azure CLI installed, you can open Create a service principal, select **Try It**.

2. Sign in to your Azure subscription.

   Azure CLI

   ```
   az login
   ```

3. First, get your subscription ID. In the console, enter the following command:

   Azure CLI

   ```
   az account show --query id -o tsv
   ```

4. Create a resource group for your security principal, specifying a location and name. This example uses the West US region.

   Azure CLI

   ```
   az group create -l westus -n MyResourceGroup
   ```

5. Create the service principal, replacing the placeholder values with valid values. You'll need a descriptive security principal name, subscription ID, resource group name.

   Notice that the security principal has "owner" permissions, necessary for creating or updating an Azure resource. If you're managing an existing search service, use contributor or "Search Service Contributor" (quote enclosed) instead.

   Azure CLI

   ```
   az ad sp create-for-rbac --name mySecurityPrincipalName \
                            --role owner \
                            --scopes
   /subscriptions/mySubscriptionID/resourceGroups/myResourceGroupName
   ```

   A successful response includes "appId", "password", and "tenant". You'll use these values for the variables "clientId", "clientSecret", and "tenant" in the next section.

Set up Postman

The following steps are from this blog post if you need more detail.

1. Start a new Postman collection and edit its properties. In the Variables tab, create the following variables:

Variable	Description
clientId	Provide the previously generated "appID" that you created in Azure AD.
clientSecret	Provide the "password" that was created for your client.
tenantId	Provide the "tenant" that was returned in the previous step.
subscriptionId	Provide the subscription ID for your subscription.
resource	Enter `https://management.azure.com/`. This Azure resource is used for all control plane operations.
bearerToken	(leave blank; the token is generated programmatically)

2. In the Authorization tab, select **Bearer Token** as the type.

3. In the **Token** field, specify the variable placeholder `{{bearerToken}}`.

4. In the Pre-request Script tab, paste in the following script:

JavaScript

```
pm.test("Check for collectionVariables", function () {
    let vars = ['clientId', 'clientSecret', 'tenantId',
'subscriptionId'];
    vars.forEach(function (item, index, array) {
        console.log(item, index);
        pm.expect(pm.collectionVariables.get(item), item + " variable
not set").to.not.be.undefined;
        pm.expect(pm.collectionVariables.get(item), item + " variable
not set").to.not.be.empty;
    });

    if (!pm.collectionVariables.get("bearerToken") || Date.now() > new
Date(pm.collectionVariables.get("bearerTokenExpiresOn") * 1000)) {
        pm.sendRequest({
            url: 'https://login.microsoftonline.com/' +
pm.collectionVariables.get("tenantId") + '/oauth2/token',
            method: 'POST',
            header: 'Content-Type: application/x-www-form-urlencoded',
            body: {
                mode: 'urlencoded',
                urlencoded: [
```

```
                    { key: "grant_type", value: "client_credentials",
disabled: false },
                    { key: "client_id", value:
pm.collectionVariables.get("clientId"), disabled: false },
                    { key: "client_secret", value:
pm.collectionVariables.get("clientSecret"), disabled: false },
                    { key: "resource", value:
pm.collectionVariables.get("resource") ||
"https://management.azure.com/", disabled: false }
                ]
            }
        }, function (err, res) {
            if (err) {
                console.log(err);
            } else {
                let resJson = res.json();
                pm.collectionVariables.set("bearerTokenExpiresOn",
resJson.expires_on);
                pm.collectionVariables.set("bearerToken",
resJson.access_token);
            }
        });
    }
});
```

5. Save the collection.

Now that Postman is set up, you can send REST calls similar to the ones described in this article. You'll update the endpoint and request body where applicable.

List search services

Returns all search services under the current subscription, including detailed service information:

HTTP

```
GET
https://management.azure.com/subscriptions/{{subscriptionId}}/providers/Micr
osoft.Search/searchServices?api-version=2022-09-01
```

Create or update a service

Creates or updates a search service under the current subscription. This example uses variables for the search service name and region, which haven't been defined yet. Either provide the names directly, or add new variables to the collection.

```HTTP
PUT
https://management.azure.com/subscriptions/{{subscriptionId}}/resourceGroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2022-09-01
{
  "location": "{{region}}",
  "sku": {
    "name": "basic"
  },
  "properties": {
    "replicaCount": 1,
    "partitionCount": 1,
    "hostingMode": "default"
  }
}
```

Create an S3HD service

To create an S3HD service, use a combination of `-Sku` and `-HostingMode` properties. Set "sku" to `Standard3` and "hostingMode" to `HighDensity`.

```HTTP
PUT
https://management.azure.com/subscriptions/{{subscriptionId}}/resourceGroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2022-09-01
{
  "location": "{{region}}",
  "sku": {
    "name": "Standard3"
  },
  "properties": {
    "replicaCount": 1,
    "partitionCount": 1,
    "hostingMode": "HighDensity"
  }
}
```

Configure role-based access for data plane

Applies to: Search Index Data Contributor, Search Index Data Reader, Search Service Contributor

In this step, configure your search service to recognize an **authorization** header on data requests that provide an OAuth2 access token.

To use Azure role-based access control (Azure RBAC) for data plane operations, set "authOptions" to "aadOrApiKey" and then send the request.

If you want to use Azure RBAC exclusively, turn off API key authentication by following up with a second request, this time setting "disableLocalAuth" to "true".

HTTP

```
PATCH
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2022-09-01
{
  "properties": {
    "disableLocalAuth": false,
    "authOptions": {
      "aadOrApiKey": {
        "aadAuthFailureMode": "http401WithBearerChallenge"
      }
    }
  }
}
```

(preview) Enforce a customer-managed key policy

If you're using customer-managed encryption, you can enable "encryptionWithCMK" with "enforcement" set to "Enabled" if you want the search service to report its compliance status.

When you enable this policy, any REST calls that create objects containing sensitive data, such as the connection string within a data source, will fail if an encryption key isn't provided: `"Error creating Data Source: "CannotCreateNonEncryptedResource: The creation of non-encrypted DataSources is not allowed when encryption policy is enforced."`

HTTP

```
PATCH
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2021-04-01-preview
{
```

```
  "properties": {
    "encryptionWithCmk": {
      "enforcement": "Enabled",
      "encryptionComplianceStatus": "Compliant"
    },
  }
}
```

(preview) Disable semantic search

Although semantic search isn't enabled by default, you could lock down the feature at the service level.

HTTP

```
PATCH
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2021-04-01-Preview
{
  "properties": {
    "semanticSearch": "disabled"
  }
}
```

(preview) Disable workloads that push data to external resources

Azure Cognitive Search writes to external data sources when updating a knowledge store, saving debug session state, or caching enrichments. The following example disables these workloads at the service level.

HTTP

```
PATCH
https://management.azure.com/subscriptions/{{subscriptionId}}/resourcegroups
/{{resource-group}}/providers/Microsoft.Search/searchServices/{{search-
service-name}}?api-version=2021-04-01-preview
{
  "properties": {
    "disabledDataExfiltrationOptions": [
      "All"
    ]
  }
}
```

Next steps

After a search service is configured, next steps include create an index or query an index using the portal, REST APIs, or the .NET SDK.

- Create an Azure Cognitive Search index in the Azure portal
- Set up an indexer to load data from other services
- Query an Azure Cognitive Search index using Search explorer in the Azure portal
- How to use Azure Cognitive Search in .NET

Move your Azure Cognitive Search service to another Azure region

Article • 01/31/2023 • 2 minutes to read

Occasionally, customers ask about moving a search service to another region. Currently, there is no built-in mechanism or tooling to help with that task, but this article can help you understand the manual steps for recreating indexes and other objects on a new search service in a different region.

> **Note**
>
> In the Azure portal, all services have an **Export template** command. In the case of Azure Cognitive Search, this command produces a basic definition of a service (name, location, tier, replica, and partition count), but does not recognize the content of your service, nor does it carry over keys, roles, or logs. Although the command exists, we don't recommend using it for moving a search service.

Prerequisites

- Ensure that the services and features that your account uses are supported in the target region.
- For preview features, ensure that your subscription is approved for the target region.

Prepare and move

1. Identify dependencies and related services to understand the full impact of relocating a service, in case you need to move more than just Azure Cognitive Search.

   Azure Storage is used for logging, creating a knowledge store, and is a commonly used external data source for AI enrichment and indexing. Cognitive Services is a dependency in AI enrichment. Both Cognitive Services and your search service are required to be in the same region if you are using AI enrichment.

2. Create an inventory of all objects on the service so that you know what to move: indexes, synonym maps, indexers, data sources, skillsets. If you enabled logging, create and archive any reports you might need for a historical record.

3. Check pricing and availability in the new region to ensure availability of Azure Cognitive Search plus any related services in the new region. The majority of features are available in all regions, but some preview features have restricted availability.

4. Create a service in the new region and republish from source code any existing indexes, synonym maps, indexers, data sources, and skillsets. Remember that service names must be unique so you cannot reuse the existing name. Check each skillset to see if connections to Cognitive Services are still valid in terms of the same-region requirement. Also, if knowledge stores are created, check the connection strings for Azure Storage if you are using a different service.

5. Reload indexes and knowledge stores, if applicable. You'll either use application code to push JSON data into an index, or rerun indexers to pull documents in from external sources.

6. Enable logging, and if you are using them, re-create security roles.

7. Update client applications and test suites to use the new service name and API keys, and test all applications.

Discard or clean up

Delete the old service once the new service is fully tested and operational. Deleting the service automatically deletes all content associated with the service.

Next steps

The following links can help you locate more information when completing the steps outlined above.

- Azure Cognitive Search pricing and regions
- Choose a tier
- Create a search service
- Load search documents
- Enable logging

Additional resources

Documentation

Service limits for tiers and skus - Azure Cognitive Search

Service limits used for capacity planning and maximum limits on requests and responses for Azure Cognitive Search.

Cognitive Search FAQ - Azure Cognitive Search

Get answers to common questions about Microsoft Azure Cognitive Search service, a cloud hosted search service on Microsoft Azure.

Availability and continuity - Azure Cognitive Search

learn how to make a search service highly available and resilient against period disruptions or even catastrophic failures.

Create Indexer (Azure Cognitive Search REST API-version=2020-06-30)

Indexers are resources that automate many aspects of data ingestion into an Azure Cognitive Search indexes. You must use a supported Azure data source to use this API.

Import data into a search index using Azure portal - Azure Cognitive Search

Learn about the Import Data wizard in the Azure portal used to create and load an index, and optionally invoke AI enrichment using built-in skills for natural language processing, translation, OCR, and image analysis.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Create a search service in the portal - Azure Cognitive Search

Learn how to set up an Azure Cognitive Search resource in the Azure portal. Choose resource groups, regions, and the SKU or pricing tier.

Plan and manage costs - Azure Cognitive Search

Learn about billable events, the billing model, and tips for cost control when running a Cognitive Search service.

Show 5 more

Training

Learning paths and modules

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Learning certificate

Microsoft Certified: Azure AI Fundamentals - Certifications

Azure AI Fundamentals validates foundational knowledge of machine learning and artificial intelligence concepts and related Microsoft Azure services.

Connect to Cognitive Search using key authentication

Article • 05/14/2023

Cognitive Search offers key-based authentication that you can use on connections to your search service. An API key is a unique string composed of 52 randomly generated numbers and letters. A request made to a search service endpoint will be accepted if both the request and the API key are valid.

> **Note**
>
> A quick note about how "key" terminology is used in Cognitive Search. An "API key", which is described in this article, refers to a GUID used for authenticating a request. A separate term, "document key", refers to a unique string in your indexed content that's used to uniquely identify documents in a search index.

Types of API keys

There are two kinds of keys used for authenticating a request:

Type	Permission level	Maximum	How created
Admin	Full access (read-write) for all content operations	2 [1]	Two admin keys, referred to as *primary* and *secondary* keys in the portal, are generated when the service is created and can be individually regenerated on demand.
Query	Read-only access, scoped to the documents collection of a search index	50	One query key is generated with the service. More can be created on demand by a search service administrator.

[1] Having two allows you to roll over one key while using the second key for continued access to the service.

Visually, there's no distinction between an admin key or query key. Both keys are strings composed of 52 randomly generated alpha-numeric characters. If you lose track of what type of key is specified in your application, you can check the key values in the portal.

Use API keys on connections

API keys are used for data plane (content) requests, such as creating or accessing an index or any other request that's represented in the Search REST APIs. Upon service creation, an API key is the only authentication mechanism for data plane operations, but you can replace or supplement key authentication with Azure roles if you can't use hard-coded keys in your code.

API keys are specified on client requests to a search service. Passing a valid API key on the request is considered proof that the request is from an authorized client. If you're creating, modifying, or deleting objects, you'll need an admin API key. Otherwise, query keys are typically distributed to client applications that issue queries.

You can specify API keys in a request header for REST API calls, or in code that calls the azure.search.documents client libraries in the Azure SDKs. If you're using the Azure portal to perform tasks, your role assignment determines the level of access.

Best practices for using hard-coded keys in source files include:

- During early development and proof-of-concept testing when security is looser, use sample or public data.
- For mature solutions or production scenarios, switch to Azure Active Directory and role-based access to eliminate the need for having hard-coded keys. Or, if you want to continue using API keys, be sure to always monitor who has access to your API keys and regenerate API keys on a regular cadence.

Portal

Key authentication is built in so no action is required. By default, the portal uses API keys to authenticate the request automatically. However, if you disable API keys and set up role assignments, the portal uses role assignments instead.

In Cognitive Search, most tasks can be performed in Azure portal, including object creation, indexing through the Import data wizard, and queries through Search explorer.

Note

It's considered a poor security practice to pass sensitive data such as an `api-key` in the request URI. For this reason, Azure Cognitive Search only accepts a query key as an `api-key` in the query string. As a general rule, we recommend passing your `api-key` as a request header.

Permissions to view or manage API keys

Permissions for viewing and managing API keys are conveyed through role assignments. Members of the following roles can view and regenerate keys:

- Owner
- Contributor
- Search Service Contributor
- Administrator and co-administrator (classic)

The following roles don't have access to API keys:

- Reader
- Search Index Data Contributor
- Search Index Data Reader

Find existing keys

You can view and manage API keys in the Azure portal, or through PowerShell, Azure CLI, or REST API.

Portal

1. Sign in to the Azure portal and find your search service.
2. Under **Settings**, select **Keys** to view admin and query keys.

Create query keys

Query keys are used for read-only access to documents within an index for operations targeting a documents collection. Search, filter, and suggestion queries are all operations that take a query key. Any read-only operation that returns system data or object definitions, such as an index definition or indexer status, requires an admin key.

Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Portal

1. Sign in to the Azure portal and find your search service.
2. Under **Settings**, select **Keys** to view API keys.
3. Under **Manage query keys**, use the query key already generated for your service, or create new query keys. The default query key isn't named, but other generated query keys can be named for manageability.

Regenerate admin keys

Two admin keys are created for each service so that you can rotate a primary key while using the secondary key for business continuity.

1. Under **Settings**, select **Keys**, then copy the secondary key.
2. For all applications, update the API key settings to use the secondary key.
3. Regenerate the primary key.
4. Update all applications to use the new primary key.

If you inadvertently regenerate both keys at the same time, all client requests using those keys will fail with HTTP 403 Forbidden. However, content isn't deleted and you aren't locked out permanently.

You can still access the service through the portal or programmatically. Management functions are operative through a subscription ID not a service API key, and are thus still available even if your API keys aren't.

After you create new keys via portal or management layer, access is restored to your content (indexes, indexers, data sources, synonym maps) once you provide those keys on requests.

Secure API keys

Use role assignments to restrict access to API keys.

Note that it's not possible to use customer-managed key encryption to encrypt API keys. Only sensitive data within the search service itself (for example, index content or connection strings in data source object definitions) can be CMK-encrypted.

1. Navigate to your search service page in Azure portal.
2. On the left navigation pane, select **Access control (IAM)**, and then select the **Role assignments** tab.
3. In the **Role** filter, select the roles that have permission to view or manage keys (Owner, Contributor, Search Service Contributor). The resulting security principals assigned to those roles have key permissions on your search service.
4. As a precaution, also check the **Classic administrators** tab to determine whether administrators and co-administrators have access.

See also

- Security in Azure Cognitive Search
- Azure role-based access control in Azure Cognitive Search
- Manage using PowerShell

Connect to Azure Cognitive Search using Azure role-based access control (Azure RBAC)

Article • 05/15/2023

Azure provides a global role-based access control authorization system for all services running on the platform. In Cognitive Search, you can use Azure roles for:

- Control plane operations (service administration tasks through Azure Resource Manager).
- Data plane operations, such as creating, loading, and querying indexes.

Per-user access over search results (sometimes referred to as row-level security or document-level security) isn't supported. As a workaround, create security filters that trim results by user identity, removing documents for which the requestor shouldn't have access.

> **Note**
>
> In Cognitive Search, "control plane" refers to operations supported in the **Management REST API** or equivalent client libraries. The "data plane" refers to operations against the search service endpoint, such as indexing or queries, or any other operation specified in the **Search REST API** or equivalent client libraries.

Built-in roles used in Search

The following roles are built in. If these roles are insufficient, create a custom role.

Role	Plane	Description
Owner	Control & Data	Full access to the control plane of the search resource, including the ability to assign Azure roles. Only the Owner role can enable or disable authentication options or manage roles for other users. Subscription administrators are members by default. On the data plane, this role has the same access as the Search Service Contributor role. It includes access to all data plane actions except the ability to query or index documents.

Role	Plane	Description
Contributor	Control & Data	Same level of control plane access as Owner, minus the ability to assign roles or change authentication options. On the data plane, this role has the same access as the Search Service Contributor role. It includes access to all data plane actions except the ability to query or index documents.
Reader	Control & Data	Read access across the entire service, including search metrics, content metrics (storage consumed, number of objects), and the object definitions of data plane resources (indexes, indexers, and so on). However, it can't read API keys or read content within indexes.
Search Service Contributor	Control & Data	Read-write access to object definitions (indexes, synonym maps, indexers, data sources, and skillsets). See Microsoft.Search/searchServices/* for the permissions list. This role can't access content in an index, so no querying or indexing, but it can create, delete, and list indexes, return index definitions and statistics, and test analyzers. This role is for search service administrators who need to manage the search service and its objects, but without content access.
Search Index Data Contributor	Data	Read-write access to content in all indexes on the search service. This role is for developers or index owners who need to import, refresh, or query the documents collection of an index.
Search Index Data Reader	Data	Read-only access to all search indexes on the search service. This role is for apps and users who run queries.

> **Note**
>
> If you disable Azure role-based access, built-in roles for the control plane (Owner, Contributor, Reader) continue to be available. Disabling Azure RBAC removes just the data-related permissions associated with those roles. In a disabled-RBAC scenario, Search Service Contributor is equivalent to control-plane Contributor.

Limitations

- Adoption of role-based access control might increase the latency of some requests. Each unique combination of service resource (index, indexer, etc.) and service principal used on a request triggers an authorization check. These authorization checks can add up to 200 milliseconds of latency to a request.

- In rare cases where requests originate from a high number of different service principals, all targeting different service resources (indexes, indexers, etc.), it's possible for the authorization checks to result in throttling. Throttling would only happen if hundreds of unique combinations of search service resource and service principal were used within a second.

Configure role-based access for data plane

Applies to: Search Index Data Contributor, Search Index Data Reader, Search Service Contributor

In this step, configure your search service to recognize an **authorization** header on data requests that provide an OAuth2 access token.

Azure portal

1. Sign in to Azure portal and open the search service page.

2. Select **Keys** in the left navigation pane.

3. Choose an **API access control** option. We recommend **Both** if you want flexibility or need to migrate apps.

Option	Description
API Key	(default). Requires an admin or query API keys on the request header for authorization. No roles are used.

Option	Description
Role-based access control	Requires membership in a role assignment to complete the task, described in the next step. It also requires an authorization header.
Both	Requests are valid using either an API key or role-based access control.

The change is effective immediately, but wait a few seconds before testing.

All network calls for search service operations and content respect the option you select: API keys, bearer token, or either one if you select **Both**.

When you enable role-based access control in the portal, the failure mode is "http401WithBearerChallenge" if authorization fails.

Assign roles

Role assignments are cumulative and pervasive across all tools and client libraries. You can assign roles using any of the supported approaches described in Azure role-based access control documentation.

You must be an **Owner** or have Microsoft.Authorization/roleAssignments/write permissions to manage role assignments.

Azure portal

Role assignments in the portal are service-wide. If you want to grant permissions to a single index, use PowerShell or the Azure CLI instead.

1. Open the Azure portal.
2. Navigate to your search service.
3. Select **Access Control (IAM)** in the left navigation pane.
4. Select **+ Add** > **Add role assignment**.

5. Select an applicable role:

    - Owner
    - Contributor
    - Reader
    - Search Service Contributor
    - Search Index Data Contributor
    - Search Index Data Reader

6. On the **Members** tab, select the Azure AD user or group identity.

7. On the **Review + assign** tab, select **Review + assign** to assign the role.

Test role assignments

Use a client to test role assignments. Remember that roles are cumulative and inherited roles that are scoped to the subscription or resource group can't be deleted or denied at the resource (search service) level.

Make sure that you register your client application with Azure Active Directory and have role assignments in place before testing access.

Azure portal

1. Open the Azure portal.

2. Navigate to your search service.

3. On the Overview page, select the **Indexes** tab:

- Contributors can view and create any object, but can't query an index using Search Explorer.
- Search Index Data Readers can use Search Explorer to query the index. You can use any API version to check for access. You should be able to send queries and view results, but you shouldn't be able to view the index definition.
- Search Index Data Contributors can select **New Index** to create a new index. Saving a new index verifies write access on the service.

Test as current user

If you're already a Contributor or Owner of your search service, you can present a bearer token for your user identity for authentication to Azure Cognitive Search. The following instructions explain how to set up a Postman collection to send requests as the current user.

1. Get a bearer token for the current user:

   Azure CLI

   ```
   az account get-access-token https://search.azure.com/.default
   ```

2. Start a new Postman collection and edit its properties. In the **Variables** tab, create the following variable:

Variable	Description
bearerToken	(copy-paste from get-access-token output on the command line)

3. In the Authorization tab, select **Bearer Token** as the type.
4. In the **Token** field, specify the variable placeholder `{{bearerToken}}`.
5. Save the collection.
6. Send a request to confirm access. Here's one that queries the hotels-quickstart index:

   HTTP

```
POST https://<service-name>.search.windows.net/indexes/hotels-quickstart/docs/search?api-version=2020-06-30
{
 "queryType": "simple",
 "search": "motel",
 "filter": "",
 "select": "HotelName,Description,Category,Tags",
 "count": true
 }
```

Grant access to a single index

In some scenarios, you may want to limit application's access to a single resource, such as an index.

The portal doesn't currently support role assignments at this level of granularity, but it can be done with PowerShell or the Azure CLI.

In PowerShell, use New-AzRoleAssignment, providing the Azure user or group name, and the scope of the assignment.

1. Load the Azure and AzureAD modules and connect to your Azure account:

PowerShell

```
Import-Module -Name Az
Import-Module -Name AzureAD
Connect-AzAccount
```

2. Add a role assignment scoped to an individual index:

PowerShell

```
New-AzRoleAssignment -ObjectId <objectId> `
    -RoleDefinitionName "Search Index Data Contributor" `
    -Scope  "/subscriptions/<subscription>/resourceGroups/<resource-group>/providers/Microsoft.Search/searchServices/<search-service>/indexes/<index-name>"
```

Create a custom role

If built-in roles don't provide the right combination of permissions, you can create a custom role to support the operations you require

This example clones **Search Index Data Reader** and then adds the ability to list indexes by name. Normally, listing the indexes on a search service is considered an administrative right.

Azure portal

These steps are derived from Create or update Azure custom roles using the Azure portal. Cloning from an existing role is supported in a search service page.

These steps create a custom role that augments search query rights to include listing indexes by name. Typically, listing indexes is considered an admin function.

1. In the Azure portal, navigate to your search service.
2. In the left-navigation pane, select **Access Control (IAM)**.
3. In the action bar, select **Roles**.
4. Right-click **Search Index Data Reader** (or another role) and select **Clone** to open the **Create a custom role** wizard.
5. On the Basics tab, provide a name for the custom role, such as "Search Index Data Explorer", and then select **Next**.
6. On the Permissions tab, select **Add permission**.
7. On the Add permissions tab, search for and then select the **Microsoft Search** tile.
8. Set the permissions for your custom role. At the top of the page, using the default **Actions** selection:
   - Under Microsoft.Search/operations, select **Read : List all available operations**.
   - Under Microsoft.Search/searchServices/indexes, select **Read : Read Index**.
9. On the same page, switch to **Data actions** and under Microsoft.Search/searchServices/indexes/documents, select **Read : Read Documents**.

   The JSON definition looks like the following example:

   ```
   JSON
   ```

```
{
 "properties": {
     "roleName": "search index data explorer",
     "description": "",
     "assignableScopes": [
         "/subscriptions/a5b1ca8b-bab3-4c26-aebe-
4cf7ec4791a0/resourceGroups/heidist-free-search-
svc/providers/Microsoft.Search/searchServices/demo-search-svc"
     ],
     "permissions": [
         {
             "actions": [
                 "Microsoft.Search/operations/read",
                 "Microsoft.Search/searchServices/indexes/read"
             ],
             "notActions": [],
             "dataActions": [

"Microsoft.Search/searchServices/indexes/documents/read"
             ],
             "notDataActions": []
         }
     ]
   }
 }
```

10. Select **Review + create** to create the role. You can now assign users and groups to the role.

Disable API key authentication

API keys can't be deleted, but they can be disabled on your service if you're using the Search Service Contributor, Search Index Data Contributor, and Search Index Data Reader roles and Azure AD authentication. Disabling API keys causes the search service to refuse all data-related requests that pass an API key in the header.

Owner or Contributor permissions are required to disable features.

To disable key-based authentication, use Azure portal or the Management REST API.

Portal

1. In the Azure portal, navigate to your search service.

2. In the left-navigation pane, select **Keys**.

3. Select **Role-based access control**.

The change is effective immediately, but wait a few seconds before testing. Assuming you have permission to assign roles as a member of Owner, service administrator, or co-administrator, you can use portal features to test role-based access.

Conditional Access

Conditional Access is a tool in Azure Active Directory used to enforce organizational policies. By using Conditional Access policies, you can apply the right access controls when needed to keep your organization secure. When accessing an Azure Cognitive Search service using role-based access control, Conditional Access can enforce organizational policies.

To enable a Conditional Access policy for Azure Cognitive Search, follow the below steps:

1. Sign in to the Azure portal.
2. Search for **Azure AD Conditional Access**.
3. Select **Policies**.
4. Select **+ New policy**.
5. In the **Cloud apps or actions** section of the policy, add **Azure Cognitive Search** as a cloud app depending on how you want to set up your policy.
6. Update the remaining parameters of the policy. For example, specify which users and groups this policy applies to.
7. Save the policy.

> **Important**
>
> If your search service has a managed identity assigned to it, the specific search service will show up as a cloud app that can be included or excluded as part of the Conditional Access policy. Conditional Access policies can't be enforced on a specific search service. Instead make sure you select the general **Azure Cognitive Search** cloud app.

Authorize access to a search app using Azure Active Directory

Article • 05/15/2023

Search applications that are built on Azure Cognitive Search can now use the Microsoft identity platform for authenticated and authorized access. On Azure, the identity provider is Azure Active Directory (Azure AD). A key benefit of using Azure AD is that your credentials and API keys no longer need to be stored in your code. Azure AD authenticates the security principal (a user, group, or service) running the application. If authentication succeeds, Azure AD returns the access token to the application, and the application can then use the access token to authorize requests to Azure Cognitive Search.

This article shows you how to configure your client for Azure AD:

- For authentication, you'll create a managed identity as the security principle. You could also use a different type of service principal object, but this article uses managed identities because they eliminate the need to manage credentials.
- For authorization, you'll assign an Azure role to the managed identity that grants permissions to run queries or manage indexing jobs.
- Update your client code to call TokenCredential(). For example, you can get started with new SearchClient(endpoint, new `DefaultAzureCredential()`) to authenticate via Azure AD using Azure.Identity.

Configure role-based access for data plane

Applies to: Search Index Data Contributor, Search Index Data Reader, Search Service Contributor

In this step, configure your search service to recognize an **authorization** header on data requests that provide an OAuth2 access token.

Azure portal

1. Sign in to Azure portal and open the search service page.
2. Select **Keys** in the left navigation pane.

3. Choose an **API access control** option. We recommend **Both** if you want flexibility or need to migrate apps.

Option	Description
API Key	(default) Requires an admin or query API keys on the request header for authorization. No roles are used.
Role-based access control	Requires membership in a role assignment to complete the task, described in the next step. It also requires an authorization header.
Both	Requests are valid using either an API key or role-based access control.

The change is effective immediately, but wait a few seconds before testing.

All network calls for search service operations and content will respect the option you select: API keys, bearer token, or either one if you select **Both**.

When you enable role-based access control in the portal, the failure mode will be "http401WithBearerChallenge" if authorization fails.

Create a managed identity

In this step, create a managed identity for your client application.

1. Sign in to the Azure portal.

2. Search for **Managed Identities**.

3. Select + **Create**.

4. Give your managed identity a name and select a region. Then, select **Create**.

Assign a role to the managed identity

Next, you need to grant your managed identity access to your search service. Azure Cognitive Search has various built-in roles. You can also create a custom role.

It's a best practice to grant minimum permissions. If your application only needs to handle queries, you should assign the Search Index Data Reader role. Alternatively, if it needs both read and write access on a search index, you should use the Search Index Data Contributor role.

1. Sign in to the Azure portal.

2. Navigate to your search service.

3. Select **Access control (IAM)** in the left navigation pane.

4. Select + **Add** > **Add role assignment**.

5. Select an applicable role:

   - Owner
   - Contributor
   - Reader
   - Search Service Contributor
   - Search Index Data Contributor
   - Search Index Data Reader

   For more information on the available roles, see Built-in roles used in Search.

   > **Note**
   >
   > The Owner, Contributor, Reader, and Search Service Contributor roles don't give you access to the data within a search index, so you can't query a search index or index data using those roles. For data access to a search index, you need either the Search Index Data Contributor or Search Index Data Reader role.

6. On the **Members** tab, select the managed identity that you want to give access to your search service.

7. On the **Review + assign** tab, select **Review + assign** to assign the role.

You can assign multiple roles, such as Search Service Contributor and Search Index Data Contributor, if your application needs comprehensive access to the search services, objects, and content.

You can also assign roles using PowerShell.

Set up Azure AD authentication in your client

Once you have a managed identity and a role assignment on the search service, you're ready to add code to your application to authenticate the security principal and acquire an OAuth 2.0 token.

Use the following client libraries for role-based access control:

- azure.search.documents (Azure SDK for .NET) version 11.4
- azure-search-documents (Azure SDK for Java) version 11.5.6
- azure/search-documents (Azure SDK for JavaScript) version 11.3.1
- azure.search.documents (Azure SDK for Python) version 11.3

> **Note**
>
> To learn more about the OAuth 2.0 code grant flow used by Azure AD, see **Authorize access to Azure Active Directory web applications using the OAuth 2.0 code grant flow**.

.NET SDK

The following instructions reference an existing C# sample to demonstrate the code changes.

1. As a starting point, clone the source code for the C# quickstart.

   The sample currently uses key-based authentication and the `AzureKeyCredential` to create the `SearchClient` and `SearchIndexClient` but you can make a small change to switch over to role-based authentication.

2. Update the Azure.Search.Documents Nuget package to version 11.4 or later.

3. Import the Azure.Identity library to get access to other authentication techniques.

4. Instead of using `AzureKeyCredential` in the beginning of `Main()` in Program.cs, use `DefaultAzureCredential` like in the code snippet below:

   C#

   ```
   // Create a SearchIndexClient to send create/delete index commands
   SearchIndexClient adminClient = new
   SearchIndexClient(serviceEndpoint, new DefaultAzureCredential());
   // Create a SearchClient to load and query documents
   ```

```
SearchClient srchclient = new SearchClient(serviceEndpoint,
indexName, new DefaultAzureCredential());
```

Local testing

User-assigned managed identities work only in Azure environments. If you run this code locally, `DefaultAzureCredential` will fall back to authenticating with your credentials. Make sure you've also given yourself the required access to the search service if you plan to run the code locally.

1. Verify your account has role assignments to run all of the operations in the quickstart sample. To both create and query an index, you'll need "Search Index Data Reader" and "Search Index Data Contributor".

2. Go to **Tools** > **Options** > **Azure Service Authentication** to choose your Azure sign-on account.

You should now be able to run the project from Visual Studio on your local system, using role-based access control for authorization.

> **Note**
>
> The Azure.Identity documentation has more details about `DefaultAzureCredential` and using **Azure AD authentication with the Azure SDK for .NET**. `DefaultAzureCredential` is intended to simplify getting started with the SDK by handling common scenarios with reasonable default behaviors. Developers who want more control or whose scenario isn't served by the default settings should use other credential types.

See also

- Use Azure role-based access control in Azure Cognitive Search
- Authorize access to Azure Active Directory web applications using the OAuth 2.0 code grant flow
- Integrating with Azure Active Directory
- Azure custom roles

Configure an IP firewall for Azure Cognitive Search

Article • 02/09/2023 • 3 minutes to read

Azure Cognitive Search supports IP rules for inbound access through a firewall, similar to the IP rules you'll find in an Azure virtual network security group. By applying IP rules, you can restrict search service access to an approved set of machines and cloud services. Access to data stored in your search service from the approved sets of machines and services will still require the caller to present a valid authorization token.

You can set IP rules in the Azure portal, as described in this article, or use the Management REST API, Azure PowerShell, or Azure CLI.

> **Note**
>
> To access a search service protected by an IP firewall through the portal, **allow access from a specific client and the portal IP address**.

Prerequisites

- A search service at the Basic tier or higher

Set IP ranges in Azure portal

1. Sign in to Azure portal and go to your Azure Cognitive Search service page.
2. Select **Networking** on the left navigation pane.
3. Set **Public Network Access** to **Selected Networks**. If your connectivity is set to **Disabled**, you can only access your search service via a private endpoint.

The Azure portal provides the ability to specify IP addresses and IP address ranges in the CIDR format. An example of CIDR notation is 8.8.8.0/24, which represents the IPs that range from 8.8.8.0 to 8.8.8.255.

4. Select **Add your client IP address** under **Firewall** to create an inbound rule for the IP address of your system.

5. Add other client IP addresses for other machines, devices, and services that will send requests to a search service.

After you enable the IP access control policy for your Azure Cognitive Search service, all requests to the data plane from machines outside the allowed list of IP address ranges are rejected.

Rejected requests

When requests originate from IP addresses that aren't in the allowed list, a generic **403 Forbidden** response is returned with no other details.

Allow access from the Azure portal IP address

When IP rules are configured, some features of the Azure portal are disabled. You'll be able to view and manage service level information, but portal access to indexes, indexers, and other top-level resources is restricted. You can restore portal access to the full range of search service operations by allowing access from the portal IP address and your client IP address.

To get the portal's IP address, perform `nslookup` (or `ping`) on `stamp2.ext.search.windows.net`, which is the domain of the traffic manager. For

nslookup, the IP address is visible in the "Non-authoritative answer" portion of the response.

In the following example, the IP address that you should copy is "52.252.175.48".

```bash
$ nslookup stamp2.ext.search.windows.net
Server:  ZenWiFi_ET8-0410
Address:  192.168.50.1

Non-authoritative answer:
Name:    azsyrie.northcentralus.cloudapp.azure.com
Address:  52.252.175.48
Aliases:  stamp2.ext.search.windows.net
          azs-ux-prod.trafficmanager.net
          azspncuux.management.search.windows.net
```

Services in different regions connect to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

For ping, the request will time out, but the IP address will be visible in the response. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is "52.252.175.48".

Providing IP addresses for clients ensures that the request isn't rejected outright, but for successful access to content and operations, authorization is also necessary. Use one of the following methodologies to authenticate your request:

- Key-based authentication, where an admin or query API key is provided on the request
- Role-based authorization, where the caller is a member of a security role on a search service, and the registered app presents an OAuth token from Azure Active Directory.

Next steps

If your client application is a static Web app on Azure, learn how to determine its IP range for inclusion in a search service IP firewall rule.

Inbound and outbound IP addresses in Azure App Service

Additional resources

Documentation

Connect through firewalls - Azure Cognitive Search

Configure IP firewall rules to allow data access by an Azure Cognitive Search indexer.

Availability and continuity - Azure Cognitive Search

learn how to make a search service highly available and resilient against period disruptions or even catastrophic failures.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Create a Private Endpoint for a secure connection - Azure Cognitive Search

Set up a private endpoint in a virtual network for a secure connection to an Azure Cognitive Search service.

Connect using Azure roles - Azure Cognitive Search

Use Azure role-based access control for granular permissions on service administration and content tasks.

Monitor queries - Azure Cognitive Search

Monitor query metrics for performance and throughput. Collect and analyze query string inputs in resource logs.

Configure search apps for Azure AD - Azure Cognitive Search

Acquire a token from Azure Active Directory to authorize search requests to an app built on Azure Cognitive Search.

API versions - Azure Cognitive Search

Version policy for Azure Cognitive Search REST APIs and the client library in the .NET SDK.

Show 5 more

Training

Learning paths and modules

Design an IP addressing schema for your Azure deployment - Training

In this module, you will learn how to plan and implement an IP addressing scheme for virtual networks and virtual machines in Azure.

Create a Private Endpoint for a secure connection to Azure Cognitive Search

Article • 09/15/2022 • 8 minutes to read

In this article, you'll learn how to secure an Azure Cognitive Search service so that it can't be accessed over the internet:

- Create an Azure virtual network (or use an existing one)
- Create a search service to use a private endpoint
- Create a Azure virtual machine in the same virtual network
- Connect to search using a browser session on the virtual machine

Private endpoints are provided by Azure Private Link, as a separate billable service. For more information about costs, see the pricing page.

You can create a private endpoint in the Azure portal, as described in this article. Alternatively, you can use the Management REST API version 2020-03-13, Azure PowerShell, or Azure CLI.

> **Note**
>
> Once a search service has a private endpoint, portal access to that service must be initiated from a browser session on a virtual machine inside the virtual network. See **this step** for details.

Why use a Private Endpoint for secure access?

Private Endpoints for Azure Cognitive Search allow a client on a virtual network to securely access data in a search index over a Private Link. The private endpoint uses an IP address from the virtual network address space for your search service. Network traffic between the client and the search service traverses over the virtual network and a private link on the Microsoft backbone network, eliminating exposure from the public internet. For a list of other PaaS services that support Private Link, check the availability section in the product documentation.

Private endpoints for your search service enable you to:

- Block all connections on the public endpoint for your search service.
- Increase security for the virtual network, by enabling you to block exfiltration of data from the virtual network.

- Securely connect to your search service from on-premises networks that connect to the virtual network using VPN or ExpressRoutes with private-peering.

Create the virtual network

In this section, you'll create a virtual network and subnet to host the VM that will be used to access your search service's private endpoint.

1. From the Azure portal home tab, select **Create a resource** > **Networking** > **Virtual network**.

2. In **Create virtual network**, enter or select the following values:

Setting	Value
Subscription	Select your subscription.
Resource group	Select **Create new**, enter a name, such as "myResourceGroup", then select **OK**.
Name	Enter a name, such as "MyVirtualNetwork".
Region	Select a region.

3. Accept the defaults for the rest of the settings. Select **Review + create** and then **Create**.

Create a search service with a private endpoint

In this section, you'll create a new Azure Cognitive Search service with a Private Endpoint.

1. On the upper-left side of the screen in the Azure portal, select **Create a resource** > **Web** > **Azure Cognitive Search**.

2. In **New Search Service - Basics**, enter or select the following values:

Setting	Value
PROJECT DETAILS	
Subscription	Select your subscription.
Resource group	Use the resource group that you created in the previous step.

Setting	Value
INSTANCE DETAILS	
URL	Enter a unique name.
Location	Select your region.
Pricing tier	Select **Change Pricing Tier** and choose your desired service tier. Private endpoints aren't supported on the **Free** tier. You must select **Basic** or higher.

3. Select **Next: Scale**.

4. Accept the defaults and select **Next: Networking**.

5. In **New Search Service - Networking**, select **Private** for **Endpoint connectivity(data)**.

6. Select **+ Add** under **Private endpoint**.

7. In **Create Private Endpoint**, enter or select values that associate your search service with the virtual network you created:

Setting	Value
Subscription	Select your subscription.
Resource group	Use the resource group that you created in the previous step.
Location	Select a region.
Name	Enter a name, such as "myPrivateEndpoint".
Target subresource	Accept the default **searchService**.
NETWORKING	
Virtual network	Select the virtual network you created in the previous step.
Subnet	Select the default.
PRIVATE DNS INTEGRATION	
Integrate with private DNS zone	Accept the default "Yes".
Private DNS zone	Accept the default **(New) privatelink.search.windows.net**.

8. Select **OK**.
9. Select **Review + create**. You're taken to the **Review + create** page where Azure validates your configuration.
10. When you see the **Validation passed** message, select **Create**.
11. Once provisioning of your new service is complete, browse to the resource that you created.
12. Select **Keys** from the left content menu.
13. Copy the **Primary admin key** for later, when connecting to the service.

Create a virtual machine

1. On the upper-left side of the screen in the Azure portal, select **Create a resource** > **Compute** > **Virtual machine**.
2. In **Create a virtual machine - Basics**, enter or select the following values:

Setting	Value
PROJECT DETAILS	
Subscription	Select your subscription.
Resource group	Use the resource group that you created in the previous section.
INSTANCE DETAILS	
Virtual machine name	Enter a name, such as "my-vm".
Region	Select your region.
Availability options	You can choose **No infrastructure redundancy required**, or select another option if you need the functionality.
Image	Select **Windows Server 2022 Datacenter: Azure Edition - Gen2**.
VM architecture	Accept the default **x64**.
Size	Accept the default **Standard D2S v3**.
ADMINISTRATOR ACCOUNT	

Setting	Value
Username	Enter the user name of the administrator. Use an account that's valid for your Azure subscription. You'll want to sign into Azure portal from the VM so that you can manage your search service.
Password	Enter the account password. The password must be at least 12 characters long and meet the defined complexity requirements.
Confirm Password	Reenter password.
INBOUND PORT RULES	
Public inbound ports	Accept the default **Allow selected ports**.
Select inbound ports	Accept the default **RDP (3389)**.

3. Select **Next: Disks**.

4. In **Create a virtual machine - Disks**, accept the defaults and select **Next: Networking**.

5. In **Create a virtual machine - Networking**, provide the following values:

Setting	Value
Virtual network	Select the virtual network you created in a previous step.
Subnet	Accept the default (10.1.0.0/24).
NIC network security group	Accept the default "Basic"
Public IP	Accept the default "(new) myVm-ip".
Public inbound ports	Select the default "Allow selected ports".
Select inbound ports	Select "HTTP 80", "HTTPS (443)" and "RDP (3389)".

> **Note**
>
> IPv4 addresses can be expressed in CIDR format. Remember to avoid the IP range reserved for private networking, as described in RFC 1918:
>
> - `10.0.0.0 - 10.255.255.255 (10/8 prefix)`
> - `172.16.0.0 - 172.31.255.255 (172.16/12 prefix)`

- `192.168.0.0 - 192.168.255.255 (192.168/16 prefix)`

6. Select **Review + create** for a validation check.

7. When you see the **Validation passed** message, select **Create**.

Connect to the VM

Download and then connect to the virtual machine as follows:

1. In the portal's search bar, search for the virtual machine created in the previous step.

2. Select **Connect**. After selecting the **Connect** button, **Connect to virtual machine** opens.

3. Select **Download RDP File**. Azure creates a Remote Desktop Protocol (`.rdp`) file and downloads it to your computer.

4. Open the downloaded `.rdp` file.

   a. If prompted, select **Connect**.

   b. Enter the username and password you specified when creating the VM.

   > **Note**
   >
   > You may need to select **More choices** > **Use a different account**, to specify the credentials you entered when you created the VM.

5. Select **OK**.

6. You may receive a certificate warning during the sign-in process. If you receive a certificate warning, select **Yes** or **Continue**.

7. Once the VM desktop appears, minimize it to go back to your local desktop.

Test connections

In this section, you'll verify private network access to the search service and connect privately to the using the Private Endpoint.

When the search service endpoint is private, some portal features are disabled. You'll be able to view and manage service level settings, but portal access to index data and various other components in the service, such as the index, indexer, and skillset definitions, is restricted for security reasons.

1. In the Remote Desktop of *myVM*, open PowerShell.

2. Enter `nslookup [search service name].search.windows.net`.

   You'll receive a message similar to this:

   ```
   Server:  UnKnown
   Address:  168.63.129.16
   Non-authoritative answer:
   Name:    [search service name].privatelink.search.windows.net
   Address:  10.0.0.5
   Aliases:  [search service name].search.windows.net
   ```

3. From the VM, connect to the search service and create an index. You can follow this quickstart to create a new search index in your service using the REST API. Setting up requests from a Web API test tool requires the search service endpoint (https://[search service name].search.windows.net) and the admin api-key you copied in a previous step.

4. Completing the quickstart from the VM is your confirmation that the service is fully operational.

5. Close the remote desktop connection to *myVM*.

6. To verify that your service isn't accessible on a public endpoint, open Postman on your local workstation and attempt the first several tasks in the quickstart. If you receive an error that the remote server doesn't exist, you've successfully configured a private endpoint for your search service.

Use the Azure portal to access a private search service

When the search service endpoint is private, some portal features are disabled. You can view and manage service level information, but index, indexer, and skillset information is hidden for security reasons.

To work around this restriction, connect to Azure portal from a browser on a virtual machine inside the virtual network. The portal uses the private endpoint on the connection and gives you visibility into content and operations.

1. Follow the steps to provision a VM that can access the search service through a private endpoint.

2. On a virtual machine in your virtual network, open a browser and sign into the Azure portal. The portal will use the private endpoint attached to the virtual machine to connect to your search service.

Clean up resources

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money.

You can delete individual resources or the resource group to delete everything you created in this exercise. Select the resource group on any resource's overview page, and then select **Delete**.

Next steps

In this article, you created a VM on a virtual network and a search service with a Private Endpoint. You connected to the VM from the internet and securely communicated to the search service using Private Link. To learn more about Private Endpoint, see What is Azure Private Endpoint?.

Troubleshoot issues with Shared Private Links in Azure Cognitive Search

Article • 03/14/2023 • 10 minutes to read

A shared private link allows Azure Cognitive Search to make secure outbound connections over a private endpoint when accessing customer resources in a virtual network. This article can help you resolve errors that might occur.

Creating a shared private link is search service control plane operation. You can create a shared private link using either the portal or a Management REST API. During provisioning, the state of the request is "Updating". After the operation completes successfully, status is "Succeeded". A private endpoint to the resource, along with any DNS zones and mappings, is created. This endpoint is used exclusively by your search service instance and is managed through Azure Cognitive Search.

Some common errors that occur during the creation phase are listed below.

Request validation failures

- Unsupported SKU: Shared private links are supported on the Basic tier and above. For indexers with skillsets, the minimum tier is Standard 2 (S2).

- Invalid name: Naming rules for a shared private link are:
  - Length should be between 1 to 60 characters
  - Alphanumeric characters
  - Names can include underscore `_`, period `.`, and hyphen `-` as long as it's not the first character in the name

- Invalid group ID: Group IDs are case-sensitive and must be one of the values from the table below:

Azure resource	Group ID	First available API version
Azure Storage - Blob (or) ADLS Gen 2	`blob`	`2020-08-01`
Azure Storage - Tables	`table`	`2020-08-01`
Azure Cosmos DB for NoSQL	`Sql`	`2020-08-01`
Azure SQL Database	`sqlServer`	`2020-08-01`
Azure Database for MySQL (preview)	`mysqlServer`	`2020-08-01-Preview`
Azure Key Vault	`vault`	`2020-08-01`
Azure Functions (preview)	`sites`	`2020-08-01-Preview`

  Resources marked with "(preview)" must be created using a preview version of the Management REST API versions.

- `privateLinkResourceId` type validation: Similar to `groupId`, Azure Cognitive Search validates that the "correct" resource type is specified in the `privateLinkResourceId`. The following are valid resource types:

Azure resource	Resource type	First available API version
Azure Storage	`Microsoft.Storage/storageAccounts`	`2020-08-01`
Azure Cosmos DB	`Microsoft.DocumentDb/databaseAccounts`	`2020-08-01`
Azure SQL Database	`Microsoft.Sql/servers`	`2020-08-01`
Azure Key Vault	`Microsoft.KeyVault/vaults`	`2020-08-01`
Azure Database for MySQL (preview)	`Microsoft.DBforMySQL/servers`	`2020-08-01-Preview`
Azure Functions (preview)	`Microsoft.Web/sites`	`2020-08-01-Preview`
Azure SQL Managed Instance (preview)	`Microsoft.Sql/managedInstance`	`2020-08-01-Preview`

  In addition, the specified `groupId` needs to be valid for the specified resource type. For example, `groupId` "blob" is valid for type "Microsoft.Storage/storageAccounts",

it can't be used with any other resource type. For a given search management API version, customers can find out the supported `groupId` and resource type details by utilizing the List supported API.

- Quota limit enforcement: Search services have quotas imposed on the distinct number of shared private link resources that can be created and the number of various target resource types that are being used (based on `groupId`). These are documented in the Shared private link resource limits section of the Azure Cognitive Search service limits page.

Deployment failures

A search service initiates the request to create a shared private link, but Azure Resource Manager performs the actual work. You can check the deployment's status in the portal or by query, and address any errors that might occur.

Shared private link resources that have failed Azure Resource Manager deployment will show up in List and Get API calls, but will have a "Provisioning State" of `Failed`. Once the reason of the Azure Resource Manager deployment failure has been ascertained, delete the `Failed` resource and re-create it after applying the appropriate resolution from the following table.

Deployment failure reason	Description	Resolution

Deployment failure reason	Description	Resolution
"LinkedAuthorizationFailed"	The error message states that the client has permission to create the shared private link on the search service, but doesn't have permission to perform action 'privateEndpointConnectionApproval/action' on the linked scope.	Re-check the private link ID in the request to make sure there are no errors or omissions in the URI. If Azure Cognitive Search and the Azure PaaS resource are in different subscriptions, and if you're using REST or a command line interface, make sure that the active Azure account is for the Azure PaaS resource. For REST clients, make sure you're not using an expired bearer token, and that the token is valid for the active subscription.
Network resource provider not registered on target resource's subscription	A private endpoint (and associated DNS mappings) is created for the target resource (Storage Account, Azure Cosmos DB, Azure SQL) via the `Microsoft.Network` resource provider (RP). If the subscription that hosts the target resource ("target subscription") isn't registered with `Microsoft.Network` RP, then the Azure Resource Manager deployment can fail.	You need to register this RP in their target subscription. You can register the resource provider using the Azure portal, PowerShell, or CLI.

Deployment failure reason	Description	Resolution
Invalid `groupId` for the target resource	When Azure Cosmos DB accounts are created, you can specify the API type for the database account. While Azure Cosmos DB offers several different API types, Azure Cognitive Search only supports "Sql" as the `groupId` for shared private link resources. When a shared private link of type "Sql" is created for a `privateLinkResourceId` pointing to a non-Sql database account, the Azure Resource Manager deployment will fail because of the `groupId` mismatch. The Azure resource ID of an Azure Cosmos DB account isn't sufficient to determine the API type that is being used. Azure Cognitive Search tries to create the private endpoint, which is then denied by Azure Cosmos DB.	You should ensure that the `privateLinkResourceId` of the specified Azure Cosmos DB resource is for a database account of "Sql" API type
Target resource not found	Existence of the target resource specified in `privateLinkResourceId` is checked only during the commencement of the Azure Resource Manager deployment. If the target resource is no longer available, then the deployment will fail.	You should ensure that the target resource is present in the specified subscription and resource group and isn't moved or deleted.
Transient/other errors	The Azure Resource Manager deployment can fail if there's an infrastructure outage or because of other unexpected reasons. This should be rare and usually indicates a transient state.	Retry creating this resource at a later time. If the problem persists, reach out to Azure Support.

Issues approving the backing private endpoint

A private endpoint is created to the target Azure resource as specified in the shared private link creation request. This is one of the final steps in the asynchronous Azure Resource Manager deployment operation, but Azure Cognitive Search needs to link the private endpoint's private IP address as part of its network configuration. Once this link is done, the `provisioningState` of the shared private link resource will go to a terminal success state `Succeeded`. Customers should only approve or deny(or in general modify the configuration of the backing private endpoint) after the state has transitioned to `Succeeded`. Modifying the private endpoint in any way before this could result in an

incomplete deployment operation and can cause the shared private link resource to end up (either immediately, or usually within a few hours) in a `Failed` state.

Search service network connectivity change stalled in an "Updating" state

Shared private links and private endpoints are used when search service **Public Network Access** is **Disabled**. Typically, changing network connectivity should succeed in a few minutes after the request has been accepted. In some circumstances, Azure Cognitive Search may take several hours to complete the connectivity change operation.

If you observe that the connectivity change operation is taking a significant amount of time, wait for a few hours. Connectivity change operations involve operations such as updating DNS records which may take longer than expected.

If **Public Network Access** is changed, existing shared private links and private endpoints may not work correctly. If existing shared private links and private endpoints stop working during a connectivity change operation, wait a few hours for the operation to complete. If they're still not working, try deleting and recreating them.

Shared private link resource stalled in an "Updating" or "Incomplete" state

Typically, a shared private link resource should go a terminal state (`Succeeded` or `Failed`) in a few minutes after the request has been accepted.

In rare circumstances, Azure Cognitive Search can fail to correctly mark the state of the shared private link resource to a terminal state (`Succeeded` or `Failed`). This usually occurs due to an unexpected failure. Shared private link resources are automatically transitioned to a `Failed` state if it has been "stuck" in a non-terminal state for more than a few hours.

If you observe that the shared private link resource hasn't transitioned to a terminal state, wait for a few hours to ensure that it becomes `Failed` before you can delete it and re-create it. Alternatively, instead of waiting you can try to create another shared private link resource with a different name (keeping all other parameters the same).

Updating a shared private link resource

An existing shared private link resource can be updated using the Create or Update API. Search only allows for narrow updates to the shared private link resource - only the request message can be modified via this API.

- It isn't possible to update any of the "core" properties of an existing shared private link resource (such as `privateLinkResourceId` or `groupId`) and this will always be unsupported. If any other property besides the request message needs to be changed, we advise customers to delete and re-create the shared private link resource.
- Attempt to update the request message of a shared private link resource is only possible if it has reached the provisioning state of `Succeeded`.

Deleting a shared private link resource

Customers can delete an existing shared private link resource via the Delete API. Similar to the process of creation (or update), this is also an asynchronous operation with four steps:

1. You request a search service to delete the shared private link resource.
2. The search service validates that the resource exists and is in a state valid for deletion. If so, it initiates an Azure Resource Manager delete operation to remove the resource.
3. Search queries for the completion of the operation (which usually takes a few minutes). At this point, the shared private link resource would have a provisioning state of "Deleting".
4. Once the operation completes successfully, the backing private endpoint and any associated DNS mappings are removed. The resource won't show up as part of List operation and attempting a Get operation on this resource will result in a 404 Not Found.

Some common errors that occur during the deletion phase are listed below.

Failure Type	Description	Resolution
Resource is in non-terminal state	A shared private link resource that's not in a terminal state (`Succeeded` or `Failed`) can't be deleted. It's possible (rare) for a shared private link resource to be stuck in a non-terminal state for up to 8 hours.	Wait until the resource has reached a terminal state and retry the delete request.
Delete operation failed with error "Conflict"	The Azure Resource Manager operation to delete a shared private link resource reaches out to the resource provider of the target resource specified in `privateLinkResourceId` ("target RP") before it can remove the private endpoint and DNS mappings. Customers can utilize Azure resource locks to prevent any changes to their resources. When Azure Resource Manager reaches out to the target RP, it requires the target RP to modify the state of the target resource (to remove details about the private endpoint from its metadata). When the target resource has a lock configured on it (or its resource group/subscription), the Azure Resource Manager operation fails with a "Conflict" (and appropriate details). The shared private link resource won't be deleted.	Customers should remove the lock on the target resource before retrying the deletion operation. **Note**: This problem can also occur when customers try to delete a search service with shared private link resources that point to "locked" target resources
Delete operation failed	The asynchronous Azure Resource Manager delete operation can fail in rare cases. When this operation fails, querying the state of the asynchronous operation will present customers with an error message and appropriate details.	Retry the operation at a later time, or reach out to Azure Support if the problem persists.

Failure Type	Description	Resolution
Resource stuck in "Deleting" state	In rare cases, a shared private link resource might be stuck in "Deleting" state for up to 8 hours, likely due to some catastrophic failure on the search RP.	Wait for 8 hours, after which the resource would transition to `Failed` state and then reissue the request.

Next steps

Learn more about shared private link resources and how to use it for secure access to protected content.

- Accessing protected content via indexers
- REST API reference

Additional resources

🕮 Documentation

az search

Connect through a private endpoint - Azure Cognitive Search

Configure indexer connections to access content from other Azure resources that are protected through a private endpoint.

Configure an IP firewall - Azure Cognitive Search

Configure IP control policies to restrict access to your Azure Cognitive Search service to specific IP addresses.

az search service

Use an Azure Resource Manager template to deploy a private endpoint for a web app - Azure App Service

Learn how to use ARM template to deploy a private endpoint for your web app.

az network private-link-service connection

az network private-endpoint

Create a Private Endpoint for a secure connection - Azure Cognitive Search

Set up a private endpoint in a virtual network for a secure connection to an Azure Cognitive Search service.

Show 5 more

Training

Module

Introduction to Azure Private Link - Training

Describe how Azure Private Link enables private connectivity to Azure services, including its features, how it works, and its use cases.

Make indexer connections to Azure Storage as a trusted service

Article • 12/08/2022 • 2 minutes to read

In Azure Cognitive Search, indexers that access Azure blobs can use the trusted service exception to securely access data. This mechanism offers customers who are unable to grant indexer access using IP firewall rules a simple, secure, and free alternative for accessing data in storage accounts.

> **Note**
>
> If Azure Storage is behind a firewall and in the same region as Azure Cognitive Search, you won't be able to create an inbound rule that admits requests from your search service. The solution for this scenario is for search to connect as a trusted service, as described in this article.

Prerequisites

- A search service with a system-assigned managed identity (see below).
- A storage account with the **Allow trusted Microsoft services to access this storage account** network option (see below).
- An Azure role assignment in Azure Storage that grants permissions to the search service system-assigned managed identity (see below).

> **Note**
>
> In Cognitive Search, a trusted service connection is limited to blobs and ADLS Gen2 on Azure Storage. It's unsupported for indexer connections to Azure Table Storage and Azure File Storage.
>
> A trusted service connection must use a system managed identity. A user-assigned managed identity isn't currently supported for this scenario.

Check service identity

1. Sign in to Azure portal and find your search service.

2. On the **Identity** page, make sure that a system assigned identity is enabled. Remember that user-assigned managed identities, currently in preview, won't work for a trusted service connection.

Check network settings

1. Sign in to Azure portal and find your storage account.

2. In the left navigation pane under **Security + networking**, select **Networking**.

3. On the **Firewalls and virtual networks** tab, allow access from **Selected networks**.

4. Scroll down to the **Exceptions** section.

5. Make sure the checkbox is selected for **Allow Azure services on the trusted services list to access this storage account**.

   This option will only permit the specific search service instance with appropriate role-based access to the storage account (strong authentication) to access data in the storage account, even if it's secured by IP firewall rules.

Check permissions

A system managed identity is an Azure AD login. The assignment needs **Storage Blob Data Reader** at a minimum.

1. In the left navigation pane under **Access Control**, view all role assignments and make sure that **Storage Blob Data Reader** is assigned to the search service system identity.

2. Add **Storage Blob Data Contributor** if write access is required.

   Features that require write access include enrichment caching, debug sessions, and knowledge store.

Set up and test the connection

The easiest way to test the connection is by running the Import data wizard.

1. Start the Import data wizard, selecting the Azure Blob Storage or Azure Data Lake Storage Gen2.

2. Choose a connection to your storage account, and then select **System-assigned**. Select **Next** to invoke a connection. If the index schema is detected, the connection succeeded.

See also

- Connect to other Azure resources using a managed identity
- Azure Blob indexer
- Azure Data Lake Storage Gen2 indexer
- Authenticate with Azure Active Directory
- About managed identities (Azure Active Directory)

Connect a search service to other Azure resources using a managed identity

Article • 12/16/2022 • 8 minutes to read

You can configure an Azure Cognitive Search service to connect to other Azure resources using a system-assigned or user-assigned managed identity and an Azure role assignment. Managed identities and role assignments eliminate the need for passing secrets and credentials in a connection string or code.

Prerequisites

- A search service at the Basic tier or above.
- An Azure resource that accepts incoming requests from an Azure Active Directory login that has a valid role assignment.

Supported scenarios

Cognitive Search can use a system-assigned or user-assigned managed identity on outbound connections to Azure resources. A system managed identity is indicated when a connection string is the unique resource ID of an Azure AD-aware service or application. A user-assigned managed identity is specified through an "identity" property.

A search service uses Azure Storage as an indexer data source and as a data sink for debug sessions, enrichment caching, and knowledge store. For search features that write back to storage, the managed identity needs a contributor role assignment as described in the "Assign a role" section.

Scenario	System managed identity	User-assigned managed identity (preview)
Indexer connections to supported Azure data sources [1]	Yes	Yes
Azure Key Vault for customer-managed keys	Yes	Yes
Debug sessions (hosted in Azure Storage) [1]	Yes	No

Scenario	System managed identity	User-assigned managed identity (preview)
Enrichment cache (hosted in Azure Storage) [1, 2]	Yes	Yes
Knowledge Store (hosted in Azure Storage) [1]	Yes	Yes
Custom skills (hosted in Azure Functions or equivalent)	Yes	Yes

[1] For connectivity between search and storage, your network security configuration imposes constraints on which type of managed identity you can use. Only a system managed identity can be used for a same-region connection to storage via the trusted service exception or resource instance rule. See Access to a network-protected storage account for details.

[2] One method for specifying an enrichment cache is in the Import data wizard. Currently, the wizard doesn't accept a managed identity connection string for enrichment cache. However, after the wizard completes, you can update the connection string in the indexer JSON definition to specify either a system or user-assigned managed identity, and then rerun the indexer.

Create a system managed identity

When a system-assigned managed identity is enabled, Azure creates an identity for your search service that can be used to authenticate to other Azure services within the same tenant and subscription. You can then use this identity in Azure role-based access control (Azure RBAC) assignments that allow access to data during indexing.

A system-assigned managed identity is unique to your search service and bound to the service for its lifetime.

Azure portal

1. Sign in to Azure portal and find your search service.
2. Under **Settings**, select **Identity**.
3. On the **System assigned** tab, under **Status**, select **On**.
4. Select **Save**.

After saving, you'll see an object identifier that's been assigned to your search service.

Create a user-assigned managed identity (preview)

A user-assigned managed identity is a resource on Azure. It's useful if you need more granularity in role assignments because you can create separate identities for different applications and scenarios.

> **Important**
>
> This feature is in public preview under **supplemental terms of use**. User-assigned managed identities aren't currently supported for connections to a network-

protected storage account. The search request currently requires a public IP address.

Azure portal

1. Sign in to Azure portal
2. Select + **Create a resource**.
3. In the "Search services and marketplace" search bar, search for "User Assigned Managed Identity" and then select **Create**.

4. Select the subscription, resource group, and region. Give the identity a descriptive name.
5. Select **Create** and wait for the resource to finish deploying.

   In the next several steps, you'll assign the user-assigned managed identity to your search service.

6. In your search service page, under **Settings**, select **Identity**.
7. On the **User assigned** tab, select **Add**.
8. Choose the subscription and then select the user-assigned managed resource that you created in the previous step.

Allow firewall access

If your Azure resource is behind a firewall, make sure there's an inbound rule that admits requests from your search service.

- For same-region connections to Azure Blob Storage or Azure Data Lake Storage Gen2, use a system managed identity and the trusted service exception. Optionally, you can configure a resource instance rule to admit requests.
- For all other resources and connections, configure an IP firewall rule that admits requests from Search. See Indexer access to content protected by Azure network security features for details.

Assign a role

A managed identity must be paired with an Azure role that determines permissions on the Azure resource.

- Data reader permissions are needed for indexer data connections and for accessing a customer-managed key in Azure Key Vault.
- Contributor (write) permissions are needed for AI enrichment features that use Azure Storage for hosting debug session data, enrichment caching, and long-term content storage in a knowledge store.

The following steps are for Azure Storage. If your resource is Azure Cosmos DB or Azure SQL, the steps are similar.

1. Sign in to Azure portal and find your Azure resource to which the search service must have access.
2. In Azure Storage, select **Access control (AIM)** on the left navigation pane.
3. Select **Add role assignment**.
4. On the **Role** page, select the roles needed for your search service:

Task	Role assignment
Blob indexing using an indexer	Add **Storage Blob Data Reader**
ADLS Gen2 indexing using an indexer	Add **Storage Blob Data Reader**
Table indexing using an indexer	Add **Reader and Data Access**

Task	Role assignment
File indexing using an indexer	Add **Reader and Data Access**
Write to a knowledge store	Add **Storage Blob DataContributor** for object and file projections, and **Reader and Data Access** for table projections.
Write to an enrichment cache	Add **Storage Blob Data Contributor**
Save debug session state	Add **Storage Blob Data Contributor**

5. On the **Members** page, select **Managed Identity**.

6. Select members. In the **Select managed identity** page, choose your subscription and then filter by service type, and then select the service. Only those services that have a managed identity will be available to select.

7. Select **Review + assign**.

Connection string examples

Once a managed identity is defined for the search service and given a role assignment, outbound connections can be modified to use the unique resource ID of the other Azure resource. Here are some examples of connection strings for various scenarios.

Blob data source (system):

An indexer data source includes a "credentials" property that determines how the connection is made to the data source. The following example shows a connection string specifying the unique resource ID of a storage account. Azure AD will authenticate the request using the system managed identity of the search service. Notice that the connection string doesn't include a container. In a data source definition, a container name is specified in the "container" property (not shown), not the connection string.

JSON

```
"credentials": {
    "connectionString": "ResourceId=/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"
    }
```

Blob data source (user):

A search request to Azure Storage can also be made under a user-assigned managed identity, currently in preview. The search service user identity is specified in the "identity" property. You can use either the portal or the REST API preview version 2021-04-30-Preview to set the identity.

JSON

```
"credentials": {
    "connectionString": "ResourceId=/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"
    },
  . . .
"identity": {
    "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
    "userAssignedIdentity": "/subscriptions/{subscription-
ID}/resourceGroups/{resource-group-
name}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{user-
assigned-managed-identity-name}"
  }
```

Knowledge store:

A knowledge store definition includes a connection string to Azure Storage. On Azure Storage, a knowledge store will create projections as blobs and tables. The connection string is the unique resource ID of your storage account. Notice that the string doesn't include containers or tables in the path. These are defined in the embedded projection definition, not the connection string.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "ResourceId=/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.Storage/storageAccounts/storage-account-name};"
}
```

Enrichment cache:

An indexer creates, uses, and remembers the container used for the cached enrichments. It's not necessary to include the container in the cache connection string. You can find the object ID on the **Identity** page of your search service in the portal.

JSON

```
"cache": {
  "enableReprocessing": true,
  "storageConnectionString": "ResourceId=/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name};"
}
```

Debug session:

A debug session runs in the portal and takes a connection string when you start the session. You can paste a string similar to the following example.

JSON

```
"ResourceId=/subscriptions/{subscription-ID}/resourceGroups/{resource-group-name}/providers/Microsoft.Storage/storageAccounts/{storage-account-name}/{container-name};",
```

Custom skill:

A custom skill targets the endpoint of an Azure function or app hosting custom code. The endpoint is specified in the custom skill definition. The presence of the

"authResourceId" tells the search service to connect using a managed identity, passing the application ID of the target function or app in the property.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
  "description": "A custom skill that can identify positions of different
phrases in the source text",
  "uri": "https://contoso.count-things.com",
  "authResourceId": "<Azure-AD-registered-application-ID>",
  "batchSize": 4,
  "context": "/document",
  "inputs": [ ... ],
  "outputs": [ ...]
}
```

See also

- Security overview
- AI enrichment overview
- Indexers overview
- Authenticate with Azure Active Directory
- About managed identities (Azure Active Directory)

Configure IP firewall rules to allow indexer connections from Azure Cognitive Search

Article • 10/12/2022 • 4 minutes to read

On behalf of an indexer, a search service will issue outbound calls to an external Azure resource to pull in data during indexing. If your Azure resource uses IP firewall rules to filter incoming calls, you'll need to create an inbound rule in your firewall that admits indexer requests.

This article explains how to find the IP address of your search service and configure an inbound IP rule on an Azure Storage account. While specific to Azure Storage, this approach also works for other Azure resources that use IP firewall rules for data access, such as Azure Cosmos DB and Azure SQL.

> **Note**
>
> A storage account and your search service must be in different regions if you want to define IP firewall rules. If your setup doesn't permit this, try the **trusted service exception** or **resource instance rule** instead.

Get a search service IP address

1. Get the fully qualified domain name (FQDN) of your search service. This will look like `<search-service-name>.search.windows.net`. You can find the FQDN by looking up your search service on the Azure portal.

2. Look up the IP address of the search service by performing a `nslookup` (or a `ping`) of the FQDN on a command prompt. Make sure you remove the "https://" prefix from the FQDN.

3. Copy the IP address so that you can specify it on an inbound rule in the next step. In the example below, the IP address that you should copy is "150.0.0.1".

```bash
nslookup contoso.search.windows.net
Server:  server.example.org
Address:  10.50.10.50

Non-authoritative answer:
Name:    <name>
Address:  150.0.0.1
aliases:  contoso.search.windows.net
```

Allow access from your client IP address

Client applications that push indexing and query requests to the search service must be represented in an IP range. On Azure, you can generally determine the IP address by pinging the FQDN of a service (for example, `ping <your-search-service-name>.search.windows.net` will return the IP address of a search service).

Add your client IP address to allow access to the service from the Azure portal on your current computer. Navigate to the **Networking** section on the left navigation pane.

Change **Public Network Access** to **Selected networks**, and then check **Add your client IP address** under **Firewall**.

Get the Azure portal IP address

If you're using the portal or the Import Data wizard to create an indexer, you'll need an inbound rule for the portal as well.

To get the portal's IP address, perform `nslookup` (or `ping`) on `stamp2.ext.search.windows.net`, which is the domain of the traffic manager. For nslookup, the IP address is visible in the "Non-authoritative answer" portion of the response.

In the example below, the IP address that you should copy is "52.252.175.48".

Bash

```
$ nslookup stamp2.ext.search.windows.net
Server:  ZenWiFi_ET8-0410
Address:  192.168.50.1

Non-authoritative answer:
Name:    azsyrie.northcentralus.cloudapp.azure.com
Address:  52.252.175.48
Aliases:  stamp2.ext.search.windows.net
          azs-ux-prod.trafficmanager.net
          azspncuux.management.search.windows.net
```

Services in different regions connect to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

For ping, the request will time out, but the IP address will be visible in the response. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is "52.252.175.48".

Get IP addresses for "AzureCognitiveSearch" service tag

You'll also need to create an inbound rule that allows requests from the multi-tenant execution environment. This environment is managed by Microsoft and it's used to offload processing intensive jobs that could otherwise overwhelm your search service. This section explains how to get the range of IP addresses needed to create this inbound rule.

An IP address range is defined for each region that supports Azure Cognitive Search. You'll need to specify the full range to ensure the success of requests originating from the multi-tenant execution environment.

You can get this IP address range from the `AzureCognitiveSearch` service tag.

1. Use either the discovery API or the downloadable JSON file. If the search service is the Azure Public cloud, download the Azure Public JSON file.

2. Open the JSON file and search for "AzureCognitiveSearch". For a search service in WestUS2, the IP addresses for the multi-tenant indexer execution environment are:

JSON

```
{
"name": "AzureCognitiveSearch.WestUS2",
"id": "AzureCognitiveSearch.WestUS2",
"properties": {
   "changeNumber": 1,
   "region": "westus2",
   "regionId": 38,
   "platform": "Azure",
   "systemService": "AzureCognitiveSearch",
   "addressPrefixes": [
      "20.42.129.192/26",
      "40.91.93.84/32",
      "40.91.127.116/32",
      "40.91.127.241/32",
      "51.143.104.54/32",
      "51.143.104.90/32",
      "2603:1030:c06:1::180/121"
   ],
   "networkFeatures": null
```

```
  }
  },
```

3. For IP addresses have the "/32" suffix, drop the "/32" (40.91.93.84/32 becomes 40.91.93.84 in the rule definition). All other IP addresses can be used verbatim.

4. Copy all of the IP addresses for the region.

Add IP addresses to IP firewall rules

Now that you have the necessary IP addresses, you can set up the inbound rules. The easiest way to add IP address ranges to a storage account's firewall rule is through the Azure portal.

1. Locate the storage account on the portal and open **Networking** on the left navigation pane.

2. In the **Firewall and virtual networks** tab, choose **Selected networks**.

3. Add the IP addresses obtained previously in the address range and select **Save**. You should have rules for the search service, Azure portal (optional), plus all of the IP addresses for the "AzureCognitiveSearch" service tag for your region.

It can take five to ten minutes for the firewall rules to be updated, after which indexers should be able to access storage account data behind the firewall.

Next Steps

- Configure Azure Storage firewalls
- Configure an IP firewall for Azure Cosmos DB
- Configure IP firewall for Azure SQL Server

Make outbound connections through a private link

Article • 04/19/2023

This article explains how to configure private, outbound calls from Azure Cognitive Search to Azure PaaS resources that run within a virtual network.

Setting up a private connection allows Azure Cognitive Search to connect to Azure PaaS through a virtual network IP address instead of a port that's open to the internet. The object created for the connection is called a *shared private link*. On the connection, Search uses the shared private link internally to reach an Azure PaaS resource inside the network boundary.

> **Note**
>
> If you're setting up a private indexer connection to a SQL Managed Instance, see **this article** instead.

When to use a shared private link

Cognitive Search makes outbound calls to other Azure PaaS resources in the following scenarios:

- Indexer connection requests to supported data sources
- Indexer (skillset) connections to Azure Storage for caching enrichments or writing to a knowledge store
- Encryption key requests to Azure Key Vault
- Custom skill requests to Azure Functions or similar resource

In service-to-service communications, Search typically sends a request over a public internet connection. However, if your data, key vault, or function should be accessed through a private endpoint, you can create a *shared private link*.

A shared private link is:

- Created using Azure Cognitive Search tooling, APIs, or SDKs
- Approved by the Azure PaaS resource owner
- Used internally by Search on a private connection to a specific Azure resource

Only your search service can use the private links that it creates, and there can be only one shared private link created on your service for each resource and sub-resource combination.

Once you set up the private link, it's used automatically whenever Search connects to that PaaS resource. You don't need to modify the connection string or alter the client you're using to issue the requests, although the device used for the connection must connect using an authorized IP in the Azure PaaS resource's firewall.

> **Note**
>
> There are two scenarios for using **Azure Private Link** and Azure Cognitive Search together. Creating a shared private link is one scenario, relevant when an *outbound* connection to Azure PaaS requires a private connection. The second scenario is **configure search for a private *inbound* connection** from clients that run in a virtual network. While both scenarios have a dependency on Azure Private Link, they are independent. You can create a shared private link without having to configure your own search service for a private endpoint.

Limitations

When evaluating shared private links for your scenario, remember these constraints.

- Several of the resource types used in a shared private link are in preview. If you're connecting to a preview resource (Azure Database for MySQL, Azure Functions, or Azure SQL Managed Instance), use a preview version of the Management REST API to create the shared private link. These versions include `2020-08-01-preview` or `2021-04-01-preview`.
- Indexer execution must use the private execution environment that's specific to your search service. Private endpoint connections aren't supported from the multi-tenant environment. The configuration setting for this requirement is covered in this article.

Prerequisites

- An Azure Cognitive Search at the Basic tier or higher. If you're using AI enrichment and skillsets, the tier must be Standard 2 (S2) or higher. See Service limits for details.

- An Azure PaaS resource from the following list of supported resource types, configured to run in a virtual network.
- You should have a minimum of Contributor permissions on both Azure Cognitive Search and the Azure PaaS resource for which you're creating the shared private link.

> **Note**
>
> Azure Private Link is used internally, at no charge, to set up the shared private link.

Supported resource types

You can create a shared private link for the following resources.

Resource type	Sub-resource (or Group ID)
Microsoft.Storage/storageAccounts [1]	`blob`, `table`, `dfs`, `file`
Microsoft.DocumentDB/databaseAccounts [2]	`Sql`
Microsoft.Sql/servers	`sqlServer`
Microsoft.KeyVault/vaults	`vault`
Microsoft.DBforMySQL/servers (preview)	`mysqlServer`
Microsoft.Web/sites (preview) [3]	`sites`
Microsoft.Sql/managedInstances (preview) [4]	`managedInstance`

[1] If Azure Storage and Azure Cognitive Search are in the same region, the connection to storage is made over the Microsoft backbone network, which means a shared private link is redundant for this configuration. However, if you already set up a private endpoint for Azure Storage, you should also set up a shared private link or the connection is refused on the storage side. Also, if you're using multiple storage formats for various scenarios in search, make sure to create a separate shared private link for each sub-resource.

[2] The `Microsoft.DocumentDB/databaseAccounts` resource type is used for indexer connections to Azure Cosmos DB for NoSQL. The provider name and group ID are case-sensitive.

[3] The `Microsoft.Web/sites` resource type is used for App service and Azure functions. In the context of Azure Cognitive Search, an Azure function is the more likely scenario. An Azure function is commonly used for hosting the logic of a custom skill. Azure Function has Consumption, Premium and Dedicated App Service hosting plans. The App Service Environment (ASE) and Azure Kubernetes Service (AKS) aren't supported at this time.

[4] See Create a shared private link for a SQL Managed Instance for instructions.

1 - Create a shared private link

Use the Azure portal, Management REST API, the Azure CLI, or Azure PowerShell to create a shared private link.

Here are a few tips:

- Give the private link a meaningful name. In the Azure PaaS resource, a shared private link appears alongside other private endpoints. A name like "shared-private-link-for-search" can remind you how it's used.

When you complete the steps in this section, you have a shared private link that's provisioned in a pending state. **It takes several minutes to create the link**. Once it's created, the resource owner needs to approve the request before it's operational.

Azure portal

1. Sign in to Azure portal and find your search service.
2. Under **Settings** on the left navigation pane, select **Networking**.
3. On the **Shared Private Access** tab, select **+ Add Shared Private Access**.
4. Select either **Connect to an Azure resource in my directory** or **Connect to an Azure resource by resource ID or alias**.
5. If you select the first option (recommended), the portal helps you pick the appropriate Azure resource and fills in other properties, such as the group ID of the resource and the resource type.

6. If you select the second option, enter the Azure resource ID manually and choose the appropriate group ID from the list at the beginning of this article.

7. Confirm the provisioning status is "Updating".

8. Once the resource is successfully created, the provisioning state of the resource changes to "Succeeded".

Shared private link creation workflow

A `202 Accepted` response is returned on success. The process of creating an outbound private endpoint is a long-running (asynchronous) operation. It involves deploying the following resources:

- A private endpoint, allocated with a private IP address in a `"Pending"` state. The private IP address is obtained from the address space that's allocated to the virtual network of the execution environment for the search service-specific private indexer. Upon approval of the private endpoint, any communication from Azure

Cognitive Search to the Azure resource originates from the private IP address and a secure private link channel.

- A private DNS zone for the type of resource, based on the group ID. By deploying this resource, you ensure that any DNS lookup to the private resource utilizes the IP address that's associated with the private endpoint.

2 - Approve the private endpoint connection

The resource owner must approve the connection request you created. This section assumes the portal for this step, but you can also use the REST APIs of the Azure PaaS resource. Private Endpoint Connections (Storage Resource Provider) and Private Endpoint Connections (Cosmos DB Resource Provider) are two examples.

1. In the Azure portal, open the **Networking** page of the Azure PaaS resource.

2. Find the section that lists the private endpoint connections. The following example is for a storage account.

3. Select the connection, and then select **Approve**. It can take a few minutes for the status to be updated in the portal.

After the private endpoint is approved, Azure Cognitive Search creates the necessary DNS zone mappings in the DNS zone that's created for it.

3 - Check shared private link status

On the Azure Cognitive Search side, you can confirm request approval by revisiting the Shared Private Access tab of the search service **Networking** page. Connection state should be approved.

Alternatively, you can also obtain connection state by using the GET Shared Private Link API.

.NET CLI

```
az rest --method get --uri
https://management.azure.com/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/contoso/providers/Microsoft.Search/searchService
s/contoso-search/sharedPrivateLinkResources/blob-pe?api-version=2022-09-01
```

This would return a JSON, where the connection state shows up as "status" under the "properties" section. Following is an example for a storage account.

JSON

```
{
      "name": "blob-pe",
      "properties": {
        "privateLinkResourceId": "/subscriptions/00000000-0000-0000-0000-
000000000000/resourceGroups/contoso/providers/Microsoft.Storage/storageAccou
nts/contoso-storage",
        "groupId": "blob",
        "requestMessage": "please approve",
        "status": "Approved",
        "resourceRegion": null,
        "provisioningState": "Succeeded"
      }
}
```

If the provisioning state (`properties.provisioningState`) of the resource is "Succeeded" and connection state(`properties.status`) is "Approved", it means that the shared private link resource is functional and the indexer can be configured to communicate over the private endpoint.

4 - Configure the indexer to run in the private environment

Indexer execution occurs in either a private environment that's specific to the search service, or a multi-tenant environment that's used internally to offload expensive skillset processing for multiple customers.

The execution environment is usually transparent, but once you start building firewall rules or establishing private connections, you have to take indexer execution into account. For a private connection, configure indexer execution to always run in the private environment.

This step shows you how to configure the indexer to run in the private environment using the REST API. You can also set the execution environment using the JSON editor in the portal.

> **Note**
>
> You can perform this step before the private endpoint connection is approved. However, until the private endpoint connection shows as approved, any existing indexer that tries to communicate with a secure resource (such as the storage account) will end up in a transient failure state and new indexers will fail to be created.

1. Create the data source definition, index, and skillset (if you're using one) as you would normally. There are no properties in any of these definitions that vary when using a shared private endpoint.

2. Create an indexer that points to the data source, index, and skillset that you created in the preceding step. In addition, force the indexer to run in the private execution environment by setting the indexer `executionEnvironment` configuration property to `private`.

   JSON

   ```
   {
       "name": "indexer",
       "dataSourceName": "blob-datasource",
       "targetIndexName": "index",
       "parameters": {
           "configuration": {
               "executionEnvironment": "private"
           }
       },
       "fieldMappings": []
   }
   ```

   Following is an example of the request in Postman.

PUT https://contoso-search.search.windows.net/indexers/indexer?api-version=2021-04-30

Params Authorization Headers (10) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1  {
2      "name": "indexer",
3      "dataSourceName": "blob-datasource",
4      "targetIndexName": "index",
5      "parameters": {
6          "configuration": {
7              "executionEnvironment": "private"
8          }
9      },
10     "fieldMappings": []
11 }
```

Response

After the indexer is created successfully, it should connect to the Azure resource over the private endpoint connection. You can monitor the status of the indexer by using the Indexer Status API.

> **Note**
>
> If you already have existing indexers, you can update them via the **PUT API** by setting the `executionEnvironment` to `private` or using the JSON editor in the portal.

5 - Test the shared private link

1. If you haven't done so already, verify that your Azure PaaS resource refuses connections from the public internet. If connections are accepted, review the DNS settings in the **Networking** page of your Azure PaaS resource.

2. Choose a tool that can invoke an outbound request scenario, such as an indexer connection to a private endpoint. An easy choice is using the **Import data** wizard, but you can also try the Postman app and REST APIs for more precision. Assuming that your search service isn't also configured for a private connection, the REST client connection to Search can be over the public internet.

3. Set the connection string to the private Azure PaaS resource. The format of the connection string doesn't change for shared private link. The search service invokes the shared private link internally.

For indexer workloads, the connection string is in the data source definition. An example of a data source might look like this:

```HTTP
 {
   "name": "my-blob-ds",
   "type": "azureblob",
   "subtype": null,
   "credentials": {
     "connectionString": "DefaultEndpointsProtocol=https;AccountName=
<YOUR-STORAGE-ACCOUNT>;AccountKey=..."
   }
```

4. For indexer workloads, remember to set the execution environment in the indexer definition. An example of an indexer definition might look like this:

```HTTP
"name": "indexer",
"dataSourceName": "my-blob-ds",
"targetIndexName": "my-index",
"parameters": {
   "configuration": {
       "executionEnvironment": "private"
       }
   },
"fieldMappings": []
}
```

5. Run the indexer. If the indexer execution succeeds and the search index is populated, the shared private link is working.

Troubleshooting

- If your indexer creation fails with "Data source credentials are invalid," check the approval status of the shared private link before debugging the connection. If the status is `Approved`, check the `properties.provisioningState` property. If it's `Incomplete`, there might be a problem with underlying dependencies. In this case, reissue the `PUT` request to re-create the shared private link. You might also need to repeat the approval step.

- If indexers fail consistently or intermittently, check the executionEnvironment property on the indexer. The value should be set to `private`. If you didn't set this property, and indexer runs succeeded in the past, it's because the search service

used a private environment of its own accord. A search service moves processing out of the standard environment if the system is under load.

- If you get an error when creating a shared private link, check service limits to verify that you're under the quota for your tier.

Next steps

Learn more about private endpoints and other secure connection methods:

- Troubleshoot issues with shared private link resources
- What are private endpoints?
- DNS configurations needed for private endpoints
- Indexer access to content protected by Azure network security features

Create a shared private link for a SQL Managed Instance from Azure Cognitive Search

Article • 04/19/2023

This article explains how to configure an indexer in Azure Cognitive Search for a private connection to a SQL Managed Instance that runs within a virtual network.

On a private connection to a SQL Managed Instance, the fully qualified domain name (FQDN) of the instance must include the DNS Zone. Currently, only the Azure Cognitive Search Management REST API provides a `resourceRegion` parameter for accepting the DNS zone specification.

Although you can call the Management REST API directly, it's easier to use the Azure CLI `az rest` module to send Management REST API calls from a command line. This article uses the Azure CLI with REST to set up the private link.

> **Note**
>
> This article refers to Azure portal for obtaining properties and confirming steps. However, when creating the shared private link for SQL Managed Instance, make sure you're using the REST API. Although the Networking tab lists `Microsoft.Sql/managedInstances` as an option, the portal doesn't currently support the extended URL format used by SQL Managed Instance.

Prerequisites

- Azure CLI
- Azure Cognitive Search, Basic or higher. If you're using AI enrichment and skillsets, use Standard 2 (S2) or higher. See Service limits for details.
- Azure SQL Managed Instance, configured to run in a virtual network.
- You should have a minimum of Contributor permissions on both Azure Cognitive Search and SQL Managed Instance.

> **Note**

Azure Private Link is used internally, at no charge, to set up the shared private link.

1 - Retrieve connection information

Retrieve the FQDN of the managed instance, including the DNS zone. The DNS zone is part of the domain name of the SQL Managed Instance. For example, if the FQDN of the SQL Managed Instance is `my-sql-managed-instance.a1b22c333d44.database.windows.net`, the DNS zone is `a1b22c333d44`.

1. In Azure portal, find the SQL managed instance object.
2. On the **Overview** tab, locate the Host property. Copy the DNS zone portion of the FQDN for the next step.
3. On the **Connection strings** tab, copy the ADO.NET connection string for a later step. It's needed for the data source connection when testing the private connection.

For more information about connection properties, see Create an Azure SQL Managed Instance.

2 - Create the body of the request

1. Using a text editor, create the JSON for the shared private link.

JSON

```
{
    "name": "{{shared-private-link-name}}",
    "properties": {
        "privateLinkResourceId": "/subscriptions/{{target-resource-
subscription-ID}}/resourceGroups/{{target-resource-
rg}}/providers/Microsoft.Sql/managedInstances/{{target-resource-
name}}",
        "resourceRegion": "a1b22c333d44",
        "groupId": "managedInstance",
        "requestMessage": "please approve",
    }
}
```

2. Provide a meaningful name for the shared private link. The shared private link appears alongside other private endpoints. A name like "shared-private-link-for-search" can remind you how it's used.

3. Paste in the DNS zone name in "resourceRegion" that you retrieved in an earlier step.

4. Edit the "privateLinkResourceId" to reflect the private endpoint of your managed instance. Provide the subscription ID, resource group name, and object name of the managed instance.

5. Save the file locally as *create-pe.json* (or use another name, remembering to update the Azure CLI syntax in the next step).

6. In the Azure CLI, type `dir` to note the current location of the file.

3 - Create a shared private link

1. From the command line, sign into Azure using `az login`.

2. If you have multiple subscriptions, make sure you're using the one you intend to use: `az account show`.

   To set the subscription, use `az account set --subscription {{subscription ID}}`

3. Call the `az rest` command to use the Management REST API of Azure Cognitive Search.

   Because shared private link support for SQL managed instances is still in preview, you need a preview version of the REST API. Use `2021-04-01-preview` for this step`.

   Azure CLI

   ```
   az rest --method put --uri
   https://management.azure.com/subscriptions/{{search-service-
   subscription-ID}}/resourceGroups/{{search service-resource-
   group}}/providers/Microsoft.Search/searchServices/{{search-service-
   name}}/sharedPrivateLinkResources/{{shared-private-link-name}}?api-
   version=2021-04-01-preview --body @create-pe.json
   ```

   Provide the subscription ID, resource group name, and service name of your Cognitive Search resource.

   Provide the same shared private link name that you specified in the JSON body.

   Provide a path to the *create-pe.json* file if you've navigated away from the file location. You can type `dir` at the command line to confirm the file is in the current directory.

4. Press Enter to run the command.

When you complete these steps, you should have a shared private link that's provisioned in a pending state. **It takes several minutes to create the link**. Once it's created, the resource owner needs to approve the request before it's operational.

4 - Approve the private endpoint connection

On the SQL Managed Instance side, the resource owner must approve the private connection request you created.

1. In the Azure portal, open the **Private endpoint connections** tab of the managed instance.

2. Find the section that lists the private endpoint connections.

3. Select the connection, and then select **Approve**. It can take a few minutes for the status to be updated in the portal.

After the private endpoint is approved, Azure Cognitive Search creates the necessary DNS zone mappings in the DNS zone that's created for it.

5 - Check shared private link status

On the Azure Cognitive Search side, you can confirm request approval by revisiting the Shared Private Access tab of the search service **Networking** page. Connection state should be approved.

6 - Configure the indexer to run in the private environment

You can now configure an indexer and its data source to use an outbound private connection to your managed instance.

You could use the **Import data** wizard for this step, but the indexer that's generated won't be valid for this scenario. You'll need to modify the indexer JSON property as described in this step to make it compliant for this scenario. You'll then need to reset and rerun the indexer to fully test the pipeline using the updated indexer.

This article assumes Postman or equivalent tool, and uses the REST APIs to make it easier to see all of the properties. Recall that REST API calls for indexers and data sources use the Search REST APIs, not the Management REST APIs used to create the shared private link. The syntax and API versions are different between the two REST APIs.

1. Create the data source definition as you would normally for Azure SQL. Although the format of the connection string is different, the data source type and other properties are valid for SQL Managed Instance.

   Provide the connection string that you copied earlier.

   HTTP

```
POST https://myservice.search.windows.net/datasources?api-version=2020-06-30
 Content-Type: application/json
 api-key: admin-key
 {
     "name" : "my-sql-datasource",
     "description" : "A database for testing Azure Cognitive Search indexes.",
     "type" : "azuresql",
     "credentials" : {
         "connectionString" : "Server=tcp:contoso.public.0000000000.database.windows.net,1433; Persist Security Info=false; User ID=<your user name>; Password=<your password>;MultipleActiveResultsSets=False; Encrypt=True;Connection Timeout=30;"
        },
     "container" : {
         "name" : "Name of table or view to index",
         "query" : null (not supported in the Azure SQL indexer)
         },
     "dataChangeDetectionPolicy": null,
     "dataDeletionDetectionPolicy": null,
     "encryptionKey": null,
```

```
    "identity": null
}
```

> **Note**
>
> If you're familiar with data source definitions in Cognitive Search, you'll notice that data source properties don't vary when using a shared private link. That's because Search will always use a shared private link on the connection if one exists.

2. Create the indexer definition, setting the indexer execution environment to "private".

   Indexer execution occurs in either a private environment that's specific to the search service, or a multi-tenant environment that's used internally to offload expensive skillset processing for multiple customers. **When connecting over a private endpoint, indexer execution must be private.**

   HTTP

```
 POST https://myservice.search.windows.net/indexers?api-version=2020-06-30
  Content-Type: application/json
  api-key: admin-key
    {
     "name": "indexer",
     "dataSourceName": "my-sql-datasource",
     "targetIndexName": "my-search-index",
     "parameters": {
         "configuration": {
             "executionEnvironment": "private"
         }
     },
     "fieldMappings": []
     }
```

3. Run the indexer. If the indexer execution succeeds and the search index is populated, the shared private link is working.

You can monitor the status of the indexer in Azure portal or by using the Indexer Status API.

You can use **Search explorer** in Azure portal to check the contents of the index.

7 - Test the shared private link

If you ran the indexer in the previous step and successfully indexed content from your managed instance, then the test was successful. However, if the indexer fails or there's no content in the index, you can modify your objects and repeat testing by choosing any client that can invoke an outbound request from an indexer.

An easy choice is running an indexer in Azure portal, but you can also try Postman and REST APIs for more precision. Assuming that your search service isn't also configured for a private connection, the REST client connection to Search can be over the public internet.

Here are some reminders for testing:

- If you use Postman or another web testing tool, use the Management REST API and the 2021-04-01-Preview API version to create the shared private link. Use the Search REST API and a stable API version to create and invoke indexers and data sources.
- You can use the Import data wizard to create an indexer, data source, and index. However, the generated indexer won't have the correct execution environment setting.
- You can edit data source and indexer JSON in Azure portal to change properties, including the execution environment and the connection string.
- You can reset and rerun the indexer in Azure portal. Reset is important for this scenario because it forces a full reprocessing of all documents.
- You can use Search explorer to check the contents of the index.

See also

- Make outbound connections through a private endpoint
- Indexer connections to Azure SQL Managed Instance through a public endpoint
- Index data from Azure SQL
- Management REST API
- Search REST API
- Quickstart: Get started with REST

Configure customer-managed keys for data encryption in Azure Cognitive Search

Article • 03/10/2023 • 16 minutes to read

Azure Cognitive Search automatically encrypts data at rest with service-managed keys. If more protection is needed, you can supplement default encryption with another encryption layer using keys that you create and manage in Azure Key Vault.

This article walks you through the steps of setting up customer-managed key (CMK) or "bring-your-own-key" (BYOK) encryption. Here are some points to keep in mind:

- CMK encryption is enacted on individual objects. If you require CMK across your search service, set an enforcement policy.
- CMK encryption depends on Azure Key Vault. You can create your own encryption keys and store them in a key vault, or you can use Azure Key Vault APIs to generate encryption keys.
- CMK encryption becomes operational when an object is created. You can't encrypt objects that already exist. CMK encryption occurs whenever an object is saved to disk, either data at rest for long-term storage or temporary data for short-term storage. With CMK, the disk never sees unencrypted data.

CMK encrypted objects

Objects that can be encrypted include indexes, synonym lists, indexers, data sources, and skillsets. Encryption is computationally expensive to decrypt so only sensitive content is encrypted.

Encryption is performed over the following content:

- All content within indexes and synonym lists, including descriptions.
- For indexers, data sources, and skillsets, only those fields that store connection strings, descriptions, keys, and user inputs are encrypted. For example, skillsets have Cognitive Services keys, and some skills accept user inputs, such as custom entities. In both cases, keys and user inputs into skills are encrypted.

Full double encryption

When you introduce CMK encryption, you're encrypting content twice. For the objects and fields noted in the previous section, content is first encrypted with your CMK, and secondly with the Microsoft-managed key. Content is doubly encrypted on data disks for long-term storage, and on temporary disks used for short-term storage.

Enabling CMK encryption will increase index size and degrade query performance. Based on observations to date, you can expect to see an increase of 30-60 percent in query times, although actual performance will vary depending on the index definition and types of queries. Because of the negative performance impact, we recommend that you only enable this feature on indexes that really require it.

Although double encryption is now available in all regions, support was rolled out in two phases:

- The first rollout was on August 1, 2020 and included the five regions listed below. Search services created in the following regions supported CMK for data disks, but not temporary disks:
  - West US 2
  - East US
  - South Central US
  - US Gov Virginia
  - US Gov Arizona

- The second rollout on May 13, 2021 added encryption for temporary disks and extended CMK encryption to all supported regions.

  If you're using CMK from a service created during the first rollout and you also want CMK encryption over temporary disks, you'll need to create a new search service in your region of choice and redeploy your content.

Prerequisites

The following tools and services are used in this scenario.

- Azure Cognitive Search on a billable tier (Basic or above, in any region).

- Azure Key Vault, you can create key vault using Azure portal, Azure CLI, or Azure PowerShell. Create the resource in the same subscription as Azure Cognitive Search. The key vault must have **soft-delete** and **purge protection** enabled.

- Azure Active Directory. If you don't have one, set up a new tenant.

You should have a search client that can create the encrypted object. Into this code, you'll reference a key vault key and Active Directory registration information. This code

could be a working app, or prototype code such as the C# code sample DotNetHowToEncryptionUsingCMK.

> **Tip**
>
> You can use **Postman** or **Azure PowerShell** to call REST APIs that create indexes and synonym maps that include an encryption key parameter. You can also use Azure SDKs. Portal support for adding a key to indexes or synonym maps isn't supported.

Key Vault tips

If you're new to Azure Key Vault, review this quickstart to learn about basic tasks: Set and retrieve a secret from Azure Key Vault using PowerShell. Here are some tips for using Key Vault:

- Use as many key vaults as you need. Managed keys can be in different key vaults. A search service can have multiple encrypted objects, each one encrypted with a different customer-managed encryption key, stored in different key vaults.
- Enable logging on Key Vault so that you can monitor key usage.
- Remember to follow strict procedures during routine rotation of key vault keys and Active Directory application secrets and registration. Always update all encrypted content to use new secrets and keys before deleting the old ones. If you miss this step, your content can't be decrypted.

1 - Enable purge protection

As a first step, make sure soft-delete and purge protection are enabled on the key vault. Due to the nature of encryption with customer-managed keys, no one can retrieve your data if your Azure Key Vault key is deleted.

To prevent data loss caused by accidental Key Vault key deletions, soft-delete and purge protection must be enabled on the key vault. Soft-delete is enabled by default, so you'll only encounter issues if you purposely disabled it. Purge protection isn't enabled by default, but it's required for customer-managed key encryption in Cognitive Search.

You can set both properties using the portal, PowerShell, or Azure CLI commands.

Azure portal

1. Sign in to Azure portal and open your key vault overview page.
2. On the **Overview** page under **Essentials**, enable **Soft-delete** and **Purge protection**.

2 - Create a key in Key Vault

Skip key generation if you already have a key in Azure Key Vault that you want to use, but collect the key identifier. You'll need this information when creating an encrypted object.

1. Sign in to Azure portal and open your key vault overview page.
2. Select **Keys** on the left, and then select **+ Generate/Import**.
3. In the **Create a key** pane, from the list of **Options**, choose the method that you want to use to create a key. You can **Generate** a new key, **Upload** an existing key, or use **Restore Backup** to select a backup of a key.
4. Enter a **Name** for your key, and optionally select other key properties.
5. Select **Create** to start the deployment.
6. Select the key, select the current version, and then make a note of the key identifier. It's composed of the **key value Uri**, the **key name**, and the **key version**. You'll need the identifier to define an encrypted index in Azure Cognitive Search.

3 - Create a security principal

You have several options for accessing the encryption key at run time. The simplest approach is to retrieve the key using the managed identity and permissions of your search service. You can use either a system or user-managed identity. Doing so allows you to omit the steps for application registration and application secrets, and simplifies the encryption key definition.

Alternatively, you can create and register an Azure Active Directory application. The search service will provide the application ID on requests.

A managed identity enables your search service to authenticate to Azure Key Vault without storing credentials (ApplicationID or ApplicationSecret) in code. The lifecycle of this type of managed identity is tied to the lifecycle of your search service, which can

only have one managed identity. For more information about how managed identities work, see What are managed identities for Azure resources.

System-managed identity

1. Make your search service a trusted service.

Conditions that will prevent you from adopting this approach include:

- You can't directly grant your search service access permissions to the key vault (for example, if the search service is in a different Active Directory tenant than the Azure Key Vault).
- A single search service is required to host multiple encrypted indexes or synonym maps, each using a different key from a different key vault, where each key vault must use **a different identity** for authentication. Because a search service can only have one managed identity, a requirement for multiple identities will disqualify the simplified approach for your scenario.

4 - Grant permissions

In this step, you'll create an access policy in Key Vault. This policy gives the application you registered with Active Directory permission to use your customer-managed key.

Access permissions could be revoked at any given time. Once revoked, any search service index or synonym map that uses that key vault will become unusable. Restoring key vault access permissions at a later time will restore index\synonym map access. For more information, see Secure access to a key vault.

1. Still in the Azure portal, open your key vault **Overview** page.

2. Select the **Access policies** on the left, and select + **Create** to start the **Create an access policy** wizard.

3. On the **Permissions** page, select *Get* for **Key permissions**, **Secret permissions**, and **Certificate Permissions**. Select *Unwrap Key* and *Wrap Key* for ** cryptographic operations on the key.

4. Select **Next**.

5. On the **Principle** page, find and select the security principal used by the search service to access the encryption key. This will either be the system-managed or user-managed identity of the search service, or the registered application.

6. Select **Next** and **Create**.

> ⓘ **Important**
>
> Encrypted content in Azure Cognitive Search is configured to use a specific Azure Key Vault key with a specific **version**. If you change the key or version, the index or synonym map must be updated to use it **before** you delete the previous one. Failing to do so will render the index or synonym map unusable. You won't be able to decrypt the content if the key is lost.

5 - Encrypt content

Encryption keys are added when you create an object. To add a customer-managed key on an index, synonym map, indexer, data source, or skillset, use the Search REST API or an Azure SDK to create an object that has encryption enabled. The portal doesn't allow encryption properties on object creation.

1. Call the Create APIs to specify the **encryptionKey** property:

   - Create Index
   - Create Synonym Map
   - Create Indexer
   - Create Data Source
   - Create Skillset.

2. Insert the encryptionKey construct into the object definition. This property is a first-level property, on the same level as name and description. The following [REST examples(#rest-examples) show property placement. If you're using the same vault, key, and version, you can paste in the same "encryptionKey" construct into each object definition.

   The first example shows an "encryptionKey" for a search service that connects using a managed identity:

   JSON

   ```
   {
     "encryptionKey": {
       "keyVaultUri": "https://demokeyvault.vault.azure.net",
       "keyVaultKeyName": "myEncryptionKey",
       "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660"
     }
   }
   ```

   The second example includes "accessCredentials", necessary if you registered an application in Azure AD:

   JSON

   ```
   {
     "encryptionKey": {
       "keyVaultUri": "https://demokeyvault.vault.azure.net",
       "keyVaultKeyName": "myEncryptionKey",
       "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
       "accessCredentials": {
         "applicationId": "00000000-0000-0000-0000-000000000000",
   ```

```
        "applicationSecret": "myApplicationSecret"
      }
    }
  }
```

Once you create the encrypted object on the search service, you can use it as you would any other object of its type. Encryption is transparent to the user and developer.

> **Note**
>
> None of these key vault details are considered secret and could be easily retrieved by browsing to the relevant Azure Key Vault page in Azure portal.

6 - Set up policy

Azure policies help to enforce organizational standards and to assess compliance at-scale. Azure Cognitive Search has an optional built-in policy for service-wide CMK enforcement.

In this section, you'll set the policy that defines a CMK standard for your search service. Then, you'll set up your search service to enforce this policy.

> **Note**
>
> Policy set up requires the preview **Services - Create or Update API**.

1. Navigate to the built-in policy in your web browser. Select **Assign**

2. Set up the policy scope. In the **Parameters** section, uncheck **Only show parameters...** and set **Effect** to **Deny**.

   During evaluation of the request, a request that matches a deny policy definition is marked as non-compliant. Assuming the standard for your service is CMK encryption, "deny" means that requests that *don't* specify CMK encryption are non-compliant.

3. Finish creating the policy.

4. Call the Services - Create or Update API to enable CMK policy enforcement at the service level.

HTTP

```
PATCH
https://management.azure.com/subscriptions/[subscriptionId]/resourceGroups/[
resourceGroupName]/providers/Microsoft.Search/searchServices/[serviceName]?
api-version=2021-04-01-preview

{
    "properties": {
        "encryptionWithCmk": {
            "enforcement": "Enabled",
            "encryptionComplianceStatus": "Compliant"
        }
    }
}
```

REST examples

This section shows the JSON for several objects so that you can see where to locate "encryptionKey" in an object definition.

Index encryption

The details of creating a new index via the REST API could be found at Create Index (REST API), where the only difference is specifying the encryption key details as part of the index definition:

JSON

```
{
 "name": "hotels",
 "fields": [
  {"name": "HotelId", "type": "Edm.String", "key": true, "filterable":
```

```
true},
  {"name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": true, "facetable": false},
  {"name": "Description", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"en.lucene"},
  {"name": "Description_fr", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"fr.lucene"},
  {"name": "Category", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true},
  {"name": "Tags", "type": "Collection(Edm.String)", "searchable": true,
"filterable": true, "sortable": false, "facetable": true},
  {"name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true,
"sortable": true, "facetable": true},
  {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset", "filterable":
true, "sortable": true, "facetable": true},
  {"name": "Rating", "type": "Edm.Double", "filterable": true, "sortable":
true, "facetable": true},
  {"name": "Location", "type": "Edm.GeographyPoint", "filterable": true,
"sortable": true}
 ],
  "encryptionKey": {
    "keyVaultUri": "https://demokeyvault.vault.azure.net",
    "keyVaultKeyName": "myEncryptionKey",
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
    "accessCredentials": {
      "applicationId": "00000000-0000-0000-0000-000000000000",
      "applicationSecret": "myApplicationSecret"
    }
  }
}
```

You can now send the index creation request, and then start using the index normally.

Synonym map encryption

Create an encrypted synonym map using the Create Synonym Map Azure Cognitive Search REST API. Use the "encryptionKey" property to specify which encryption key to use.

JSON

```
{
  "name" : "synonymmap1",
  "format" : "solr",
  "synonyms" : "United States, United States of America, USA\n
  Washington, Wash. => WA",
  "encryptionKey": {
    "keyVaultUri": "https://demokeyvault.vault.azure.net",
    "keyVaultKeyName": "myEncryptionKey",
```

```
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
    "accessCredentials": {
      "applicationId": "00000000-0000-0000-0000-000000000000",
      "applicationSecret": "myApplicationSecret"
    }
  }
}
```

You can now send the synonym map creation request, and then start using it normally.

Data source encryption

Create an encrypted data source using the Create Data Source (REST API). Use the "encryptionKey" property to specify which encryption key to use.

JSON

```
{
  "name" : "datasource1",
  "type" : "azureblob",
  "credentials" :
  { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=datasource;AccountKey=accountkey
;EndpointSuffix=core.windows.net"
  },
  "container" : { "name" : "containername" },
  "encryptionKey": {
    "keyVaultUri": "https://demokeyvault.vault.azure.net",
    "keyVaultKeyName": "myEncryptionKey",
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
    "accessCredentials": {
      "applicationId": "00000000-0000-0000-0000-000000000000",
      "applicationSecret": "myApplicationSecret"
    }
  }
}
```

You can now send the data source creation request, and then start using it normally.

Skillset encryption

Create an encrypted skillset using the Create Skillset REST API. Use the "encryptionKey" property to specify which encryption key to use.

JSON

```
{
    "name": "skillset1",
```

```
    "skills":  [ omitted for brevity ],
    "cognitiveServices": { omitted for brevity },
      "knowledgeStore":  { omitted for brevity  },
    "encryptionKey": (optional) {
        "keyVaultKeyName": "myEncryptionKey",
        "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
        "keyVaultUri": "https://demokeyvault.vault.azure.net",
        "accessCredentials": {
            "applicationId": "00000000-0000-0000-0000-000000000000",
            "applicationSecret": "myApplicationSecret"}
    }
}
```

You can now send the skillset creation request, and then start using it normally.

Indexer encryption

Create an encrypted indexer using the Create Indexer REST API. Use the "encryptionKey" property to specify which encryption key to use.

JSON

```
{
  "name": "indexer1",
  "dataSourceName": "datasource1",
  "skillsetName": "skillset1",
  "parameters": {
      "configuration": {
          "imageAction": "generateNormalizedImages"
      }
  },
  "encryptionKey": {
    "keyVaultUri": "https://demokeyvault.vault.azure.net",
    "keyVaultKeyName": "myEncryptionKey",
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
    "accessCredentials": {
      "applicationId": "00000000-0000-0000-0000-000000000000",
      "applicationSecret": "myApplicationSecret"
    }
  }
}
```

You can now send the indexer creation request, and then start using it normally.

> **Important**
>
> While "encryptionKey" can't be added to existing search indexes or synonym maps, it may be updated by providing different values for any of the three key vault

details (for example, updating the key version). When changing to a new Key Vault key or a new key version, any search index or synonym map that uses the key must first be updated to use the new key\version **before** deleting the previous key\version. Failing to do so will render the index or synonym map unusable, as it won't be able to decrypt the content once key access is lost. Although restoring key vault access permissions at a later time will restore content access.

Work with encrypted content

With customer-managed key encryption, you'll notice latency for both indexing and queries due to the extra encrypt/decrypt work. Azure Cognitive Search doesn't log encryption activity, but you can monitor key access through key vault logging. We recommend that you enable logging as part of key vault configuration.

Key rotation is expected to occur over time. Whenever you rotate keys, it's important to follow this sequence:

1. Determine the key used by an index or synonym map.
2. Create a new key in key vault, but leave the original key available.
3. Update the encryptionKey properties on an index or synonym map to use the new values. Only objects that were originally created with this property can be updated to use a different value.
4. Disable or delete the previous key in the key vault. Monitor key access to verify the new key is being used.

For performance reasons, the search service caches the key for up to several hours. If you disable or delete the key without providing a new one, queries will continue to work on a temporary basis until the cache expires. However, once the search service can no longer decrypt content, you'll get this message: "Access forbidden. The query key used might have been revoked - please retry."

Next steps

If you're unfamiliar with Azure security architecture, review the Azure Security documentation, and in particular, this article:

Data encryption-at-rest

Additional resources

Documentation

Configure Virtual Networks for Azure Cognitive Services - Azure Cognitive Services

Configure layered network security for your Cognitive Services resources.

Customer-Managed Keys for Cognitive Services - Cognitive Services

Learn how to use the Azure portal to configure customer-managed keys with Azure Key Vault. Customer-managed keys enable you to create, rotate, disable, and revoke access controls.

Azure Cognitive Services security - Azure Cognitive Services

Learn about the security considerations for Cognitive Services usage.

Rotate keys in Azure Cognitive Services - Azure Cognitive Services

Learn how to rotate API keys for better security, without interrupting service

Configure secure access with managed identities and private endpoints - Azure Applied AI Services

Learn how to configure secure communications between Form Recognizer and other Azure Services.

Connect using Azure roles - Azure Cognitive Search

Use Azure role-based access control for granular permissions on service administration and content tasks.

Azure Policy Regulatory Compliance controls for Azure Cognitive Services - Azure Cognitive Services

Lists Azure Policy Regulatory Compliance controls available for Azure Cognitive Services. These built-in policy definitions provide common approaches to managing the compliance of your Azure resources.

Form Recognizer service encryption of data at rest - Azure Applied AI Services

Microsoft offers Microsoft-managed encryption keys, and also lets you manage your Cognitive Services subscriptions with your own keys, called customer-managed keys (CMK). This article covers data encryption at rest for Form Recognizer, and how to enable and manage CMK.

Show 5 more

Training

Module

Encrypt public sector data in Azure - Training

Overview of the options available for managing encryption keys and encrypting data throughout its lifecycle, with special attention to public sector.

Find encrypted objects and information

Article • 03/10/2023 • 2 minutes to read

In Azure Cognitive Search, customer-managed encryption keys are created, stored, and managed in Azure Key Vault. If you need to determine whether an object is encrypted, or what key name or version was used in Azure Key Vault, use the REST API or an Azure SDK to retrieve the **encryptionKey** property from the object definition in your search service.

Objects that aren't encrypted with a customer-managed key will have an empty **encryptionKey** property. Otherwise, you might see a definition similar to the following example.

JSON

```
"encryptionKey": {
"keyVaultUri": "https://demokeyvault.vault.azure.net",
"keyVaultKeyName": "myEncryptionKey",
"keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",
"accessCredentials": {
    "applicationId": "00000000-0000-0000-0000-000000000000",
    "applicationSecret": "myApplicationSecret"
    }
}
```

The **encryptionKey** construct is the same for all encrypted objects. It's a first-level property, on the same level as the object name and description.

Get the admin API key

Before you can retrieve object definitions from a search service, you'll need to provide an admin API key. Admin API keys are required on requests that query for object definitions and metadata. The easiest way to get the admin API key is through the portal.

1. Sign in to the Azure portal and open the search service overview page.

2. On the left side, select **Keys** and copy an admin API.

For the remaining steps, switch to PowerShell and the REST API. The portal doesn't show encryption key information for any object.

Retrieve object properties

Use PowerShell and REST to run the following commands to set up the variables and get object definitions.

Alternatively, you can also use the Azure SDKs for .NET, Python, JavaScript, and Java.

First, connect to your Azure account.

PowerShell

```
Connect-AzAccount
```

Set up the headers used on each request in the current session. Provide the admin API key used for search service authentication.

PowerShell

```
$headers = @{
'api-key' = '<YOUR-ADMIN-API-KEY>'
'Content-Type' = 'application/json'
'Accept' = 'application/json' }
```

To return a list of all search indexes, set the endpoint to the indexes collection.

PowerShell

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes?api-
version=2020-06-30&$select=name'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

To return a specific index definition, provide its name in the path. The encryptionKey property is at the end.

PowerShell

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/<YOUR-INDEX-
NAME>?api-version=2020-06-30'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

To return synonym maps, set the endpoint to the synonyms collection and then send the request.

PowerShell

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms?api-version=2020-06-30&$select=name'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

The following example returns a specific synonym map definition, including the encryptionKey property at the end.

PowerShell

```
$uri= 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms/<YOUR-SYNONYM-MAP-NAME>?api-version=2020-06-30'
Invoke-RestMethod -Uri $uri -Headers $headers | ConvertTo-Json
```

Use the same pattern to return the encryptionKey property for other top-level objects such as indexers, skillsets, data sources, and index aliases.

Next steps

We recommend that you enable logging on Azure Key Vault so that you can monitor key usage.

For more information about using Azure Key or configuring customer managed encryption:

- Quickstart: Set and retrieve a secret from Azure Key Vault using PowerShell
- Configure customer-managed keys for data encryption in Azure Cognitive Search

---

Additional resources

🕮 Documentation

Configure an IP firewall - Azure Cognitive Search

Configure IP control policies to restrict access to your Azure Cognitive Search service to specific IP addresses.

Encrypt data using customer-managed keys - Azure Cognitive Search

Supplement server-side encryption in Azure Cognitive Search using customer managed keys (CMK) or bring your own keys (BYOK) that you create and manage in Azure Key Vault.

Connect using Azure roles - Azure Cognitive Search

Use Azure role-based access control for granular permissions on service administration and content tasks.

Create a Private Endpoint for a secure connection - Azure Cognitive Search

Set up a private endpoint in a virtual network for a secure connection to an Azure Cognitive Search service.

Connect through a private endpoint - Azure Cognitive Search

Configure indexer connections to access content from other Azure resources that are protected through a private endpoint.

Security overview - Azure Cognitive Search

Learn about the security features in Azure Cognitive Search to protect endpoints, content, and operations.

Preview feature list - Azure Cognitive Search

Preview features are released so that customers can provide feedback on their design and utility. This article is a comprehensive list of all features currently in preview.

Configure search apps for Azure AD - Azure Cognitive Search

Acquire a token from Azure Active Directory to authorize search requests to an app built on Azure Cognitive Search.

Show 5 more

Security filters for trimming results in Azure Cognitive Search

Article • 03/26/2023 • 5 minutes to read

Cognitive Search doesn't provide document-level permissions and can't vary search results from within the same index by user permissions. As a workaround, you can create a filter that trims search results based on a string containing a group or user identity.

This article describes a pattern for security filtering that includes following steps:

- ✓ Assemble source documents with the required content
- ✓ Create a field for the principal identifiers
- ✓ Push the documents to the search index for indexing
- ✓ Query the index with the `search.in` filter function

About the security filter pattern

Although Cognitive Search doesn't integrate with security subsystems for access to content within an index, many customers who have document-level security requirements have found that filters can meet their needs.

In Cognitive Search, a security filter is a regular OData filter that includes or excludes a search result based on a matching value, except that in a security filter, the criteria is a string consisting of a security principal. There's no authentication or authorization through the security principal. The principal is just a string, used in a filter expression, to include or exclude a document from the search results.

There are several ways to achieve security filtering. One way is through a complicated disjunction of equality expressions: for example, `Id eq 'id1' or Id eq 'id2'`, and so forth. This approach is error-prone, difficult to maintain, and in cases where the list contains hundreds or thousands of values, slows down query response time by many seconds.

A better solution is using the `search.in` function for security filters, as described in this article. If you use `search.in(Id, 'id1, id2, ...')` instead of an equality expression, you can expect subsecond response times.

Prerequisites

- The field containing group or user identity must be a string with the "filterable" attribute. It should be a collection. It shouldn't allow nulls.
- Other fields in the same document should provide the content that's accessible to that group or user. In the following JSON documents, the "security_id" fields contain identities used in a security filter, and the name, salary, and marital status will be included if the identity of the caller matches the "security_id" of the document.

JSON

```
{
    "Employee-1": {
        "id": "100-1000-10-1-10000-1",
        "name": "Abram",
        "salary": 75000,
        "married": true,
        "security_id": "10011"
    },
    "Employee-2": {
        "id": "200-2000-20-2-20000-2",
        "name": "Adams",
        "salary": 75000,
        "married": true,
        "security_id": "20022"
    }
}
```

> **Note**
>
> The process of retrieving the principal identifiers and injecting those strings into source documents that can be indexed by Cognitive Search isn't covered in this article. Refer to the documentation of your identity service provider for help with obtaining identifiers.

Create security field

In the search index, within the field collection, you need one field that contains the group or user identity, similar to the fictitious "security_id" field in the previous example.

1. Add a security field as a `Collection(Edm.String)`. Make sure it has a `filterable` attribute set to `true` so that search results are filtered based on the access the user has. For example, if you set the `group_ids` field to `["group_id1, group_id2"]` for

the document with `file_name` "secured_file_b", only users that belong to group IDs "group_id1" or "group_id2" have read access to the file.

Set the field's `retrievable` attribute to `false` so that it isn't returned as part of the search request.

2. Indexes require a document key. The "file_id" field satisfies that requirement. Indexes should also contain searchable content. The "file_name" and "file_description" fields represent that in this example.

```https
POST https://[search
service].search.windows.net/indexes/securedfiles/docs/index?api-
version=2020-06-30
{
     "name": "securedfiles",
     "fields": [
         {"name": "file_id", "type": "Edm.String", "key": true,
"searchable": false },
         {"name": "file_name", "type": "Edm.String", "searchable": true
},
         {"name": "file_description", "type": "Edm.String",
"searchable": true },
         {"name": "group_ids", "type": "Collection(Edm.String)",
"filterable": true, "retrievable": false }
     ]
 }
```

Push data into your index using the REST API

Issue an HTTP POST request to your index's URL endpoint. The body of the HTTP request is a JSON object containing the documents to be indexed:

```HTTP
POST https://[search
service].search.windows.net/indexes/securedfiles/docs/index?api-
version=2020-06-30
```

In the request body, specify the content of your documents:

```JSON
{
    "value": [
        {
```

```
            "@search.action": "upload",
            "file_id": "1",
            "file_name": "secured_file_a",
            "file_description": "File access is restricted to the Human
Resources.",
            "group_ids": ["group_id1"]
        },
        {
            "@search.action": "upload",
            "file_id": "2",
            "file_name": "secured_file_b",
            "file_description": "File access is restricted to Human
Resources and Recruiting.",
            "group_ids": ["group_id1", "group_id2"]
        },
        {
            "@search.action": "upload",
            "file_id": "3",
            "file_name": "secured_file_c",
            "file_description": "File access is restricted to Operations and
Logistics.",
            "group_ids": ["group_id5", "group_id6"]
        }
    ]
}
```

If you need to update an existing document with the list of groups, you can use the `merge` or `mergeOrUpload` action:

JSON

```
{
    "value": [
        {
            "@search.action": "mergeOrUpload",
            "file_id": "3",
            "group_ids": ["group_id7", "group_id8", "group_id9"]
        }
    ]
}
```

For more information on uploading documents, see Add, Update, or Delete Documents (REST).

Apply the security filter in the query

In order to trim documents based on `group_ids` access, you should issue a search query with a `group_ids/any(g:search.in(g, 'group_id1, group_id2,...'))` filter, where

'group_id1, group_id2,...' are the groups to which the search request issuer belongs.

This filter matches all documents for which the `group_ids` field contains one of the given identifiers. For full details on searching documents using Azure Cognitive Search, you can read Search Documents.

This sample shows how to set up query using a POST request.

Issue the HTTP POST request:

HTTP

```
POST https://[service
name].search.windows.net/indexes/securedfiles/docs/search?api-version=2020-
06-30
Content-Type: application/json
api-key: [admin or query key]
```

Specify the filter in the request body:

JSON

```
{
   "filter":"group_ids/any(g:search.in(g, 'group_id1, group_id2'))"
}
```

You should get the documents back where `group_ids` contains either "group_id1" or "group_id2". In other words, you get the documents to which the request issuer has read access.

JSON

```
{
 [
   {
    "@search.score":1.0,
     "file_id":"1",
     "file_name":"secured_file_a",
   },
   {
     "@search.score":1.0,
     "file_id":"2",
     "file_name":"secured_file_b"
   }
 ]
}
```

Next steps

This article described a pattern for filtering results based on user identity and the `search.in()` function. You can use this function to pass in principal identifiers for the requesting user to match against principal identifiers associated with each target document. When a search request is handled, the `search.in` function filters out search results for which none of the user's principals have read access. The principal identifiers can represent things like security groups, roles, or even the user's own identity.

For an alternative pattern based on Azure Active Directory, or to revisit other security features, see the following links.

- Security filters for trimming results using Active Directory identities
- Security in Azure Cognitive Search

Security filters for trimming Azure Cognitive Search results using Active Directory identities

Article • 03/24/2023 • 6 minutes to read

This article demonstrates how to use Azure Active Directory (AD) security identities together with filters in Azure Cognitive Search to trim search results based on user group membership.

This article covers the following tasks:

- ✓ Create Azure AD groups and users
- ✓ Associate the user with the group you have created
- ✓ Cache the new groups
- ✓ Index documents with associated groups
- ✓ Issue a search request with group identifiers filter

> **Note**
>
> Sample code snippets in this article are written in C#. You can find the full source code **on GitHub**.

Prerequisites

Your index in Azure Cognitive Search must have a security field to store the list of group identities having read access to the document. This use case assumes a one-to-one correspondence between a securable item (such as an individual's college application) and a security field specifying who has access to that item (admissions personnel).

You must have Azure AD administrator permissions (Owner or administrator), required in this walkthrough for creating users, groups, and associations.

Your application must also be registered with Azure AD as a multi-tenant app, as described in the following procedure.

Register your application with Azure Active Directory

This step integrates your application with Azure AD for the purpose of accepting sign-ins of user and group accounts. If you aren't a tenant admin in your organization, you

might need to create a new tenant to perform the following steps.

1. In Azure portal, find the Azure Active Directory resource for your subscription.

2. On the left, under **Manage**, select **App registrations**, and then select **New registration**.

3. Give the registration a name, perhaps a name that is similar to the search application name. Select **Register**.

4. Once the app registration is created, copy the Application ID. You'll need to provide this string to your application.

   If you're stepping through the DotNetHowToSecurityTrimming, paste this value into the **app.config** file.

   Repeat for the Tenant ID.

5. On the left, select **API permissions** and then select **Add a permission**.

6. Select **Microsoft Graph** and then select **Delegated permissions**.

7. Search for and then add the following delegated permissions:

   - **Directory.ReadWrite.All**
   - **Group.ReadWrite.All**
   - **User.ReadWrite.All**

Microsoft Graph provides an API that allows programmatic access to Azure AD through a REST API. The code sample for this walkthrough uses the permissions to call the Microsoft Graph API for creating groups, users, and associations. The APIs are also used to cache group identifiers for faster filtering.

Create users and groups

If you're adding search to an established application, you might have existing user and group identifiers in Azure AD. In this case, you can skip the next three steps.

However, if you don't have existing users, you can use Microsoft Graph APIs to create the security principals. The following code snippets demonstrate how to generate identifiers, which become data values for the security field in your Azure Cognitive Search index. In our hypothetical college admissions application, this would be the security identifiers for admissions staff.

User and group membership might be very fluid, especially in large organizations. Code that builds user and group identities should run often enough to pick up changes in organization membership. Likewise, your Azure Cognitive Search index requires a similar update schedule to reflect the current status of permitted users and resources.

Step 1: Create Group

C#

```
private static Dictionary<Group, List<User>> CreateGroupsWithUsers(string
tenant)
{
    Group group = new Group()
    {
        DisplayName = "My First Prog Group",
        SecurityEnabled = true,
        MailEnabled = false,
        MailNickname = "group1"
    };
```

Step 2: Create User

C#

```
User user1 = new User()
{
    GivenName = "First User",
    Surname = "User1",
    MailNickname = "User1",
    DisplayName = "First User",
    UserPrincipalName = String.Format("user1@{0}", tenant),
    PasswordProfile = new PasswordProfile() { Password = "********" },
    AccountEnabled = true
};
```

Step 3: Associate user and group

```C#
List<User> users = new List<User>() { user1, user2 };
Dictionary<Group, List<User>> groups = new Dictionary<Group, List<User>>() {
{ group, users } };
```

Step 4: Cache the groups identifiers

Optionally, to reduce network latency, you can cache the user-group associations so that when a search request is issued, groups are returned from the cache, saving a roundtrip to Azure AD. You can use Azure AD Batch API to send a single Http request with multiple users and build the cache.

Microsoft Graph is designed to handle a high volume of requests. If an overwhelming number of requests occur, Microsoft Graph fails the request with HTTP status code 429. For more information, see Microsoft Graph throttling.

Index document with their permitted groups

Query operations in Azure Cognitive Search are executed over an Azure Cognitive Search index. In this step, an indexing operation imports searchable data into an index, including the identifiers used as security filters.

Azure Cognitive Search doesn't authenticate user identities, or provide logic for establishing which content a user has permission to view. The use case for security trimming assumes that you provide the association between a sensitive document and the group identifier having access to that document, imported intact into a search index.

In the hypothetical example, the body of the PUT request on an Azure Cognitive Search index would include an applicant's college essay or transcript along with the group identifier having permission to view that content.

In the generic example used in the code sample for this walkthrough, the index action might look as follows:

```C#
private static void IndexDocuments(string indexName, List<string> groups)
{
    IndexDocumentsBatch<SecuredFiles> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new SecuredFiles()
            {
                FileId = "1",
```

```
                Name = "secured_file_a",
                GroupIds = new[] { groups[0] }
            },
              ...
            };

IndexDocumentsResult result = searchClient.IndexDocuments(batch);
```

Issue a search request

For security trimming purposes, the values in your security field in the index are static values used for including or excluding documents in search results. For example, if the group identifier for Admissions is "A11B22C33D44-E55F66G77-H88I99JKK", any documents in an Azure Cognitive Search index having that identifier in the security filed are included (or excluded) in the search results sent back to the requestor.

To filter documents returned in search results based on groups of the user issuing the request, review the following steps.

Step 1: Retrieve user's group identifiers

If the user's groups weren't already cached, or the cache has expired, issue the groups request.

C#

```
private static async void RefreshCache(IEnumerable<User> users)
{
    HttpClient client = new HttpClient();
    var userGroups = await _microsoftGraphHelper.GetGroupsForUsers(client,
users);
    _groupsCache = new ConcurrentDictionary<string, List<string>>
(userGroups);
}
```

Step 2: Compose the search request

Assuming you have the user's groups membership, you can issue the search request with the appropriate filter values.

C#

```
private static void SearchQueryWithFilter(string user)
{
    // Using the filter below, the search result will contain all documents
```

```
that their GroupIds field
    // contain any one of the Ids in the groups list
    string filter = String.Format("groupIds/any(p:search.in(p, '{0}'))",
string.Join(",", String.Join(",", _groupsCache[user])));
    SearchOptions searchOptions =
        new SearchOptions()
        {
            Filter = filter
        };
    searchOptions.Select.Add("name");

    SearchResults<SecuredFiles> results = searchClient.Search<SecuredFiles>
("*", searchOptions);

    Console.WriteLine("Results for groups '{0}' : {1}", _groupsCache[user],
results.GetResults().Select(r => r.Document.Name));
}
```

Step 3: Handle the results

The response includes a filtered list of documents, consisting of those that the user has permission to view. Depending on how you construct the search results page, you might want to include visual cues to reflect the filtered result set.

Next steps

In this walkthrough, you learned a pattern for using Azure AD sign-ins to filter documents in Azure Cognitive Search results, trimming the results of documents that don't match the filter provided on the request. For an alternative pattern that might be simpler, or to revisit other security features, see the following links.

- Security filters for trimming results
- Security in Azure Cognitive Search

API versions in Azure Cognitive Search

Article • 03/23/2023 • 2 minutes to read

Azure Cognitive Search rolls out feature updates regularly. Sometimes, but not always, these updates require a new version of the API to preserve backward compatibility. Publishing a new version allows you to control when and how you integrate search service updates in your code.

As a rule, the REST APIs and libraries are versioned only when necessary, since it can involve some effort to upgrade your code to use a new API version. A new version is needed only if some aspect of the API has changed in a way that breaks backward compatibility. Such changes can happen because of fixes to existing features, or because of new features that change existing API surface area.

See Azure SDK lifecycle and support policy for more information about the deprecation path.

Unsupported versions

Some API versions are discontinued and will be rejected by a search service:

- **2015-02-28**
- **2015-02-28-Preview**
- **2014-07-31-Preview**
- **2014-10-20-Preview**

All SDKs are based on REST API versions. If a REST version is discontinued, any SDK that's based on it is also discontinued. All Azure Cognitive Search .NET SDKs older than **3.0.0-rc** are now discontinued.

Support for the above-listed versions was discontinued on October 15, 2020. If you have code that uses a discontinued version, you can migrate existing code to a newer REST API version or to a newer Azure SDK.

REST APIs

REST API	Link
Search Service (data plane)	See API versions in REST API reference
Management (control plane)	See API versions in REST API reference

Azure SDK for .NET

The following table provides links to more recent SDK versions.

SDK version	Status	Description
Azure.Search.Documents 11	Active	New client library from the Azure .NET SDK team, initially released July 2020. See the Change Log for information about minor releases.
Microsoft.Azure.Search 10	Retired	Released May 2019. This is the last version of the Microsoft.Azure.Search package and it's now deprecated. It's succeeded by Azure.Search.Documents.
Microsoft.Azure.Management.Search 4.0.0	Active	Targets the Management REST api-version=2020-08-01.
Microsoft.Azure.Management.Search 3.0.0	Active	Targets the Management REST api-version=2015-08-19.

Azure SDK for Java

SDK version	Status	Description
Java azure-search-documents 11	Active	New client library from Azure Java SDK, released July 2020. Targets the Search REST api-version=2019-05-06.
Java Management Client 1.35.0	Active	Targets the Management REST api-version=2015-08-19.

Azure SDK for JavaScript

SDK version	Status	Description
JavaScript @azure/search-documents 11.0	Active	New client library from Azure JavaScript & TypesScript SDK, released July 2020. Targets the Search REST api-version=2016-09-01.
JavaScript @azure/arm-search	Active	Targets the Management REST api-version=2015-08-19.

Azure SDK for Python

SDK version	Status	Description
Python azure-search-documents 11.0	Active	New client library from Azure Python SDK, released July 2020. Targets the Search REST api-version=2019-05-06.
Python azure-mgmt-search 8.0	Active	Targets the Management REST api-version=2015-08-19.

All Azure SDKs

If you are looking for beta client libraries and documentation, this page contains links to all of the Azure SDK library packages, code, and docs.

Preview features in Azure Cognitive Search

Article • 05/05/2023

This article is a complete list of all features that are in public preview. This list is helpful if you're checking feature status.

Preview functionality is provided under Supplemental Terms of Use, without a service level agreement, and isn't recommended for production workloads.

Preview features that transition to general availability are removed from this list. If a feature isn't listed below, you can assume it's generally available or retired. For announcements regarding general availability and retirement, see Service Updates or What's New.

Feature	Category	Description	Availability
Azure Files indexer	Indexer data source	Adds REST API support for creating indexers for Azure Files	Public preview, Search REST API 2021-04-30-Preview. Announced in November 2021.
Search REST API 2021-04-30-Preview	Security	Modifies Create or Update Data Source to support managed identities under Azure Active Directory, for indexers that connect to external data sources.	Public preview, Search REST API 2021-04-30-Preview. Announced in May 2021.
Management REST API 2021-04-01-Preview	Security	Modifies Create or Update Service to support new DataPlaneAuthOptions.	Public preview, Management REST API, API version 2021-04-01-Preview. Announced in May 2021.
Reset Documents	Indexer	Reprocesses individually selected search documents in indexer workloads.	Use the Reset Documents REST API, API versions 2021-04-30-Preview or 2020-06-30-Preview.

Feature	Category	Description	Availability
SharePoint Indexer	Indexer data source	New data source for indexer-based indexing of SharePoint content.	Sign up is required so that support can be enabled for your subscription on the backend. Configure this data source using Create or Update Data Source, API versions 2021-04-30-Preview or 2020-06-30-Preview, or the Azure portal.
MySQL indexer data source	Indexer data source	Index content and metadata from Azure MySQL data sources.	Sign up is required so that support can be enabled for your subscription on the backend. Configure this data source using Create or Update Data Source, API versions 2021-04-30-Preview or 2020-06-30-Preview, .NET SDK 11.2.1, and Azure portal.
Azure Cosmos DB indexer: Azure Cosmos DB for MongoDB, Azure Cosmos DB for Apache Gremlin	Indexer data source	For Azure Cosmos DB, SQL API is generally available, but Azure Cosmos DB for MongoDB and Azure Cosmos DB for Apache Gremlin are in preview.	For MongoDB and Gremlin, sign up first so that support can be enabled for your subscription on the backend. MongoDB data sources can be configured in the portal. Configure this data source using Create or Update Data Source, API versions 2021-04-30-Preview or 2020-06-30-Preview.
Native blob soft delete	Indexer data source	The Azure Blob Storage indexer in Azure Cognitive Search will recognize blobs that are in a soft deleted state, and remove the corresponding search document during indexing.	Configure this data source using Create or Update Data Source, API versions 2021-04-30-Preview or 2020-06-30-Preview.
Semantic search	Relevance (scoring)	Semantic ranking of results, captions, and answers.	Configure semantic search using Search Documents, API versions 2021-04-30-Preview or 2020-06-30-Preview, and Search Explorer (portal).

Feature	Category	Description	Availability
speller	Query	Optional spelling correction on query term inputs for simple, full, and semantic queries.	Search Preview REST API, API versions 2021-04-30-Preview or 2020-06-30-Preview, and Search Explorer (portal).
Normalizers	Query	Normalizers provide simple text pre-processing: consistent casing, accent removal, and ASCII folding, without invoking the full text analysis chain.	Use Search Documents, API versions 2021-04-30-Preview or 2020-06-30-Preview.
featuresMode parameter	Relevance (scoring)	Relevance score expansion to include details: per field similarity score, per field term frequency, and per field number of unique tokens matched. You can consume these data points in custom scoring solutions.	Add this query parameter using Search Documents, API versions 2021-04-30-Preview, 2020-06-30-Preview, or 2019-05-06-Preview.
Azure Machine Learning (AML) skill	AI enrichment (skills)	A new skill type to integrate an inferencing endpoint from Azure Machine Learning.	Use Search Preview REST API, API versions 2021-04-30-Preview, 2020-06-30-Preview, or 2019-05-06-Preview. Also available in the portal, in skillset design, assuming Cognitive Search and Azure ML services are deployed in the same subscription.
Incremental enrichment	AI enrichment (skills)	Adds caching to an enrichment pipeline, allowing you to reuse existing output if a targeted modification, such as an update to a skillset or another object, doesn't change the content. Caching applies only to enriched documents produced by a skillset.	Add this configuration setting using Create or Update Indexer Preview REST API, API versions 2021-04-30-Preview, 2020-06-30-Preview, or 2019-05-06-Preview.

Feature	Category	Description	Availability
moreLikeThis	Query	Finds documents that are relevant to a specific document. This feature has been in earlier previews.	Add this query parameter in Search Documents Preview REST API calls, with API versions 2021-04-30-Preview, 2020-06-30-Preview, 2019-05-06-Preview, 2016-09-01-Preview, or 2017-11-11-Preview.

How to call a preview REST API

Azure Cognitive Search always pre-releases experimental features through the REST API first, then through prerelease versions of the .NET SDK.

Preview features are available for testing and experimentation, with the goal of gathering feedback on feature design and implementation. For this reason, preview features can change over time, possibly in ways that break backwards compatibility. This is in contrast to features in a GA version, which are stable and unlikely to change with the exception of small backward-compatible fixes and enhancements. Also, preview features don't always make it into a GA release.

While some preview features might be available in the portal and .NET SDK, the REST API always has preview features.

- For search operations, `2021-04-30-Preview` is the current preview version.
- For management operations, `2021-04-01-Preview` is the current preview version.

Older previews are still operational but become stale over time. If your code calls `api-version=2019-05-06-Preview` or `api-version=2016-09-01-Preview` or `api-version=2017-11-11-Preview`, those calls are still valid, but those versions won't include new features and bug fixes aren't guaranteed.

The following example syntax illustrates a call to the preview API version.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-idx/docs/search?api-version=2021-04-30-Preview
  Content-Type: application/json
  api-key: [admin key]
```

Azure Cognitive Search service is available in multiple versions and client libraries. For more information, see API versions.

Next steps

Review the Search REST Preview API reference documentation. If you encounter problems, ask us for help on Stack Overflow or contact support.

Search service REST API Reference (Preview)

How to manage concurrency in Azure Cognitive Search

Article • 01/26/2021 • 5 minutes to read

When managing Azure Cognitive Search resources such as indexes and data sources, it's important to update resources safely, especially if resources are accessed concurrently by different components of your application. When two clients concurrently update a resource without coordination, race conditions are possible. To prevent this, Azure Cognitive Search offers an *optimistic concurrency model*. There are no locks on a resource. Instead, there is an ETag for every resource that identifies the resource version so that you can formulate requests that avoid accidental overwrites.

> **Tip**
>
> Conceptual code in a **sample C# solution** explains how concurrency control works in Azure Cognitive Search. The code creates conditions that invoke concurrency control. Reading the **code fragment below** might be sufficient for most developers, but if you want to run it, edit appsettings.json to add the service name and an admin api-key. Given a service URL of `http://myservice.search.windows.net`, the service name is `myservice`.

How it works

Optimistic concurrency is implemented through access condition checks in API calls writing to indexes, indexers, data sources, skillsets, and synonymMap resources.

All resources have an *entity tag (ETag)* that provides object version information. By checking the ETag first, you can avoid concurrent updates in a typical workflow (get, modify locally, update) by ensuring the resource's ETag matches your local copy.

- The REST API uses an ETag on the request header.
- The .NET SDK sets the ETag through an accessCondition object, setting the If-Match | If-Match-None header on the resource. Objects that use ETags, such as SynonymMap.ETag and SearchIndex.ETag, have an accessCondition object.

Every time you update a resource, its ETag changes automatically. When you implement concurrency management, all you're doing is putting a precondition on the update request that requires the remote resource to have the same ETag as the copy of the

resource that you modified on the client. If a concurrent process has changed the remote resource already, the ETag will not match the precondition and the request will fail with HTTP 412. If you're using the .NET SDK, this manifests as a `CloudException` where the `IsAccessConditionFailed()` extension method returns true.

> **Note**
>
> There is only one mechanism for concurrency. It's always used regardless of which API is used for resource updates.

Use cases and sample code

The following code demonstrates accessCondition checks for key update operations:

- Fail an update if the resource no longer exists
- Fail an update if the resource version changes

Sample code from DotNetETagsExplainer program

C#

```
class Program
{
    // This sample shows how ETags work by performing conditional updates
and deletes
    // on an Azure Cognitive Search index.
    static void Main(string[] args)
    {
        IConfigurationBuilder builder = new
ConfigurationBuilder().AddJsonFile("appsettings.json");
        IConfigurationRoot configuration = builder.Build();

        SearchServiceClient serviceClient =
CreateSearchServiceClient(configuration);

        Console.WriteLine("Deleting index...\n");
        DeleteTestIndexIfExists(serviceClient);

        // Every top-level resource in Azure Cognitive Search has an
associated ETag that keeps track of which version
        // of the resource you're working on. When you first create a
resource such as an index, its ETag is
        // empty.
        Index index = DefineTestIndex();
        Console.WriteLine(
            $"Test index hasn't been created yet, so its ETag should be
blank. ETag: '{index.ETag}'");
```

```
        // Once the resource exists in Azure Cognitive Search, its ETag will
be populated. Make sure to use the object
        // returned by the SearchServiceClient! Otherwise, you will still
have the old object with the
        // blank ETag.
        Console.WriteLine("Creating index...\n");
        index = serviceClient.Indexes.Create(index);

        Console.WriteLine($"Test index created; Its ETag should be
populated. ETag: '{index.ETag}'");

        // ETags let you do some useful things you couldn't do otherwise.
For example, by using an If-Match
        // condition, we can update an index using CreateOrUpdate and be
guaranteed that the update will only
        // succeed if the index already exists.
        index.Fields.Add(new Field("name", AnalyzerName.EnMicrosoft));
        index =
            serviceClient.Indexes.CreateOrUpdate(
                index,
                accessCondition:
AccessCondition.GenerateIfExistsCondition());

        Console.WriteLine(
            $"Test index updated; Its ETag should have changed since it was
created. ETag: '{index.ETag}'");

        // More importantly, ETags protect you from concurrent updates to
the same resource. If another
        // client tries to update the resource, it will fail as long as all
clients are using the right
        // access conditions.
        Index indexForClient1 = index;
        Index indexForClient2 = serviceClient.Indexes.Get("test");

        Console.WriteLine("Simulating concurrent update. To start, both
clients see the same ETag.");
        Console.WriteLine($"Client 1 ETag: '{indexForClient1.ETag}' Client 2
ETag: '{indexForClient2.ETag}'");

        // Client 1 successfully updates the index.
        indexForClient1.Fields.Add(new Field("a", DataType.Int32));
        indexForClient1 =
            serviceClient.Indexes.CreateOrUpdate(
                indexForClient1,
                accessCondition:
AccessCondition.IfNotChanged(indexForClient1));

        Console.WriteLine($"Test index updated by client 1; ETag:
'{indexForClient1.ETag}'");

        // Client 2 tries to update the index, but fails, thanks to the ETag
check.
        try
```

```
        {
            indexForClient2.Fields.Add(new Field("b", DataType.Boolean));
            serviceClient.Indexes.CreateOrUpdate(
                indexForClient2,
                accessCondition:
AccessCondition.IfNotChanged(indexForClient2));

            Console.WriteLine("Whoops; This shouldn't happen");
            Environment.Exit(1);
        }
        catch (CloudException e) when (e.IsAccessConditionFailed())
        {
            Console.WriteLine("Client 2 failed to update the index, as
expected.");
        }

        // You can also use access conditions with Delete operations. For
example, you can implement an
        // atomic version of the DeleteTestIndexIfExists method from this
sample like this:
        Console.WriteLine("Deleting index...\n");
        serviceClient.Indexes.Delete("test", accessCondition:
AccessCondition.GenerateIfExistsCondition());

        // This is slightly better than using the Exists method since it
makes only one round trip to
        // Azure Cognitive Search instead of potentially two. It also avoids
an extra Delete request in cases where
        // the resource is deleted concurrently, but this doesn't matter
much since resource deletion in
        // Azure Cognitive Search is idempotent.

        // And we're done! Bye!
        Console.WriteLine("Complete.  Press any key to end
application...\n");
        Console.ReadKey();
    }

    private static SearchServiceClient
CreateSearchServiceClient(IConfigurationRoot configuration)
    {
        string searchServiceName = configuration["SearchServiceName"];
        string adminApiKey = configuration["SearchServiceAdminApiKey"];

        SearchServiceClient serviceClient =
            new SearchServiceClient(searchServiceName, new
SearchCredentials(adminApiKey));
        return serviceClient;
    }

    private static void DeleteTestIndexIfExists(SearchServiceClient
serviceClient)
    {
        if (serviceClient.Indexes.Exists("test"))
        {
```

```
            serviceClient.Indexes.Delete("test");
        }
    }

    private static Index DefineTestIndex() =>
        new Index()
        {
            Name = "test",
            Fields = new[] { new Field("id", DataType.String) { IsKey = true
} }
        };
    }
}
```

Design pattern

A design pattern for implementing optimistic concurrency should include a loop that retries the access condition check, a test for the access condition, and optionally retrieves an updated resource before attempting to re-apply the changes.

This code snippet illustrates the addition of a synonymMap to an index that already exists. This code is from the Synonym C# example for Azure Cognitive Search.

The snippet gets the "hotels" index, checks the object version on an update operation, throws an exception if the condition fails, and then retries the operation (up to three times), starting with index retrieval from the server to get the latest version.

C#

```
private static void EnableSynonymsInHotelsIndexSafely(SearchServiceClient
serviceClient)
{
    int MaxNumTries = 3;

    for (int i = 0; i < MaxNumTries; ++i)
    {
        try
        {
            Index index = serviceClient.Indexes.Get("hotels");
            index = AddSynonymMapsToFields(index);

            // The IfNotChanged condition ensures that the index is updated
only if the ETags match.
            serviceClient.Indexes.CreateOrUpdate(index, accessCondition:
AccessCondition.IfNotChanged(index));

            Console.WriteLine("Updated the index successfully.\n");
            break;
        }
```

```
        catch (CloudException e) when (e.IsAccessConditionFailed())
        {
            Console.WriteLine($"Index update failed : {e.Message}.
Attempt({i}/{MaxNumTries}).\n");
        }
    }
}

private static Index AddSynonymMapsToFields(Index index)
{
    index.Fields.First(f => f.Name == "category").SynonymMaps = new[] {
"desc-synonymmap" };
    index.Fields.First(f => f.Name == "tags").SynonymMaps = new[] { "desc-
synonymmap" };
    return index;
}
```

Next steps

Try modifying other samples to exercise ETags or AccessCondition objects.

- search-dotnet-getting-started on GitHub. This repository includes the "DotNetEtagsExplainer" project.
- azure-search-dotnet-samples on GitHub contains additional C# samples.

See also

- Common HTTP request and response headers
- HTTP status codes
- Index operations (REST API)

How to use Azure.Search.Documents in a C# .NET Application

Article • 01/26/2023 • 17 minutes to read

This article explains how to create and manage search objects using C# and the **Azure.Search.Documents** (version 11) client library in the Azure SDK for .NET.

About version 11

Azure SDK for .NET includes an **Azure.Search.Documents** client library from the Azure SDK team that is functionally equivalent to the previous client library, Microsoft.Azure.Search. Version 11 is more consistent in terms of Azure programmability. Some examples include AzureKeyCredential key authentication, and System.Text.Json.Serialization for JSON serialization.

As with previous versions, you can use this library to:

- Create and manage search indexes, data sources, indexers, skillsets, and synonym maps
- Load and manage search documents in an index
- Execute queries, all without having to deal with the details of HTTP and JSON
- Invoke and manage AI enrichment (skillsets) and outputs

The library is distributed as a single Azure.Search.Documents NuGet package, which includes all APIs used for programmatic access to a search service.

The client library defines classes like `SearchIndex`, `SearchField`, and `SearchDocument`, as well as operations like `SearchIndexClient.CreateIndex` and `SearchClient.Search` on the `SearchIndexClient` and `SearchClient` classes. These classes are organized into the following namespaces:

- Azure.Search.Documents
- Azure.Search.Documents.Indexes
- Azure.Search.Documents.Indexes.Models
- Azure.Search.Documents.Models

Azure.Search.Documents (version 11) targets version 2020-06-30 of the Azure Cognitive Search REST API.

The client library doesn't provide service management operations, such as creating and scaling search services and managing API keys. If you need to manage your search

resources from a .NET application, use the Microsoft.Azure.Management.Search library in the Azure SDK for .NET.

Upgrade to v11

If you have been using the previous version of the .NET SDK and you'd like to upgrade to the current generally available version, see Upgrade to Azure Cognitive Search .NET SDK version 11.

SDK requirements

- Visual Studio 2019 or later.
- Your own Azure Cognitive Search service. In order to use the SDK, you'll need the name of your service and one or more API keys. Create a service in the portal if you don't have one.
- Download the Azure.Search.Documents package using **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution** in Visual Studio. Search for the package name `Azure.Search.Documents`.

Azure SDK for .NET conforms to .NET Standard 2.0.

Example application

This article "teaches by example", relying on the DotNetHowTo code example on GitHub to illustrate fundamental concepts in Azure Cognitive Search - specifically, how to create, load, and query a search index.

For the rest of this article, assume a new index named "hotels", populated with a few documents, with several queries that match on results.

Below is the main program, showing the overall flow:

```C#
// This sample shows how to delete, create, upload documents and query an index
static void Main(string[] args)
{
    IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.json");
    IConfigurationRoot configuration = builder.Build();
```

```
    SearchIndexClient indexClient = CreateSearchIndexClient(configuration);

    string indexName = configuration["SearchIndexName"];

    Console.WriteLine("{0}", "Deleting index...\n");
    DeleteIndexIfExists(indexName, indexClient);

    Console.WriteLine("{0}", "Creating index...\n");
    CreateIndex(indexName, indexClient);

    SearchClient searchClient = indexClient.GetSearchClient(indexName);

    Console.WriteLine("{0}", "Uploading documents...\n");
    UploadDocuments(searchClient);

    SearchClient indexClientForQueries =
CreateSearchClientForQueries(indexName, configuration);

    Console.WriteLine("{0}", "Run queries...\n");
    RunQueries(indexClientForQueries);

    Console.WriteLine("{0}", "Complete.  Press any key to end
application...\n");
    Console.ReadKey();
}
```

Next is a partial screenshot of the output, assuming you run this application with a valid service name and API keys:

Client types

The client library uses three client types for various operations: SearchIndexClient to create, update, or delete indexes, SearchClient to load or query an index, and

SearchIndexerClient to work with indexers and skillsets. This article focuses on the first two.

At a minimum, all of the clients require the service name or endpoint, and an API key. It's common to provide this information in a configuration file, similar to what you find in the `appsettings.json` file of the DotNetHowTo sample application. To read from the configuration file, add `using Microsoft.Extensions.Configuration;` to your program.

The following statement creates the index client used to create, update, or delete indexes. It takes a search endpoint and admin API key.

C#

```
private static SearchIndexClient CreateSearchIndexClient(IConfigurationRoot
configuration)
{
    string searchServiceEndPoint = configuration["SearchServiceEndPoint"];
    string adminApiKey = configuration["SearchServiceAdminApiKey"];

    SearchIndexClient indexClient = new SearchIndexClient(new
Uri(searchServiceEndPoint), new AzureKeyCredential(adminApiKey));
    return indexClient;
}
```

The next statement creates the search client used to load documents or run queries. `SearchClient` requires an index. You'll need an admin API key to load documents, but you can use a query API key to run queries.

C#

```
string indexName = configuration["SearchIndexName"];

private static SearchClient CreateSearchClientForQueries(string indexName,
IConfigurationRoot configuration)
{
    string searchServiceEndPoint = configuration["SearchServiceEndPoint"];
    string queryApiKey = configuration["SearchServiceQueryApiKey"];

    SearchClient searchClient = new SearchClient(new
Uri(searchServiceEndPoint), indexName, new AzureKeyCredential(queryApiKey));
    return searchClient;
}
```

Note

If you provide an invalid key for the import operation (for example, a query key where an admin key was required), the `SearchClient` will throw a `CloudException`

with the error message "Forbidden" the first time you call an operation method on it. If this happens to you, double-check the API key.

Deleting the index

In the early stages of development, you might want to include a DeleteIndex statement to delete a work-in-progress index so that you can recreate it with an updated definition. Sample code for Azure Cognitive Search often includes a deletion step so that you can rerun the sample.

The following line calls `DeleteIndexIfExists`:

```C#
Console.WriteLine("{0}", "Deleting index...\n");
DeleteIndexIfExists(indexName, indexClient);
```

This method uses the given `SearchIndexClient` to check if the index exists, and if so, deletes it:

```C#
private static void DeleteIndexIfExists(string indexName, SearchIndexClient
indexClient)
{
    try
    {
        if (indexClient.GetIndex(indexName) != null)
        {
            indexClient.DeleteIndex(indexName);
        }
    }
    catch (RequestFailedException e) when (e.Status == 404)
    {
        // Throw an exception if the index name isn't found
        Console.WriteLine("The index doesn't exist. No deletion occurred.");
```

> **Note**
>
> The example code in this article uses the synchronous methods for simplicity, but you should use the asynchronous methods in your own applications to keep them scalable and responsive. For example, in the method above you could use **DeleteIndexAsync** instead of **DeleteIndex**.

Create an index

You can use SearchIndexClient to create an index.

The method below creates a new SearchIndex object with a list of SearchField objects that define the schema of the new index. Each field has a name, data type, and several attributes that define its search behavior.

Fields can be defined from a model class using FieldBuilder. The `FieldBuilder` class uses reflection to create a list of `SearchField` objects for the index by examining the public properties and attributes of the given `Hotel` model class. We'll take a closer look at the `Hotel` class later on.

```C#
private static void CreateIndex(string indexName, SearchIndexClient
indexClient)
{
    FieldBuilder fieldBuilder = new FieldBuilder();
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var definition = new SearchIndex(indexName, searchFields);

    indexClient.CreateOrUpdateIndex(definition);
}
```

Besides fields, you could also add scoring profiles, suggesters, or CORS options to the index (these parameters are omitted from the sample for brevity). You can find more information about the SearchIndex object and its constituent parts in the SearchIndex properties list, as well as in the REST API reference.

> ⓘ **Note**
>
> You can always create the list of `Field` objects directly instead of using `FieldBuilder` if needed. For example, you may not want to use a model class or you may need to use an existing model class that you don't want to modify by adding attributes.

Call CreateIndex in Main()

`Main` creates a new "hotels" index by calling the above method:

```C#
```

```
Console.WriteLine("{0}", "Creating index...\n");
CreateIndex(indexName, indexClient);
```

Use a model class for data representation

The DotNetHowTo sample uses model classes for the Hotel, Address, and Room data structures. `Hotel` references `Address`, a single level complex type (a multi-part field), and `Room` (a collection of multi-part fields).

You can use these types to create and load the index, and to structure the response from a query:

C#

```
// Use-case: <Hotel> in a field definition
FieldBuilder fieldBuilder = new FieldBuilder();
var searchFields = fieldBuilder.Build(typeof(Hotel));

// Use-case: <Hotel> in a response
private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.GetResults())
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}
```

An alternative approach is to add fields to an index directly. The following example shows just a few fields.

C#

```
 SearchIndex index = new SearchIndex(indexName)
 {
     Fields =
         {
             new SimpleField("hotelId", SearchFieldDataType.String) { IsKey
= true, IsFilterable = true, IsSortable = true },
             new SearchableField("hotelName") { IsFilterable = true,
IsSortable = true },
             new SearchableField("hotelCategory") { IsFilterable = true,
IsSortable = true },
             new SimpleField("baseRate", SearchFieldDataType.Int32) {
IsFilterable = true, IsSortable = true },
             new SimpleField("lastRenovationDate",
SearchFieldDataType.DateTimeOffset) { IsFilterable = true, IsSortable = true
```

```
}
        }
 };
```

Field definitions

Your data model in .NET and its corresponding index schema should support the search experience you'd like to give to your end user. Each top level object in .NET, such as a search document in a search index, corresponds to a search result you would present in your user interface. For example, in a hotel search application your end users may want to search by hotel name, features of the hotel, or the characteristics of a particular room.

Within each class, a field is defined with a data type and attributes that determine how it's used. The name of each public property in each class maps to a field with the same name in the index definition.

Take a look at the following snippet that pulls several field definitions from the Hotel class. Notice that Address and Rooms are C# types with their own class definitions (refer to the sample code if you want to view them). Both are complex types. For more information, see How to model complex types.

C#

```
public partial class Hotel
{
    [SimpleField(IsKey = true, IsFilterable = true)]
    public string HotelId { get; set; }

    [SearchableField(IsSortable = true)]
    public string HotelName { get; set; }

    [SearchableField(AnalyzerName = LexicalAnalyzerName.Values.EnLucene)]
    public string Description { get; set; }

    [SearchableField(IsFilterable = true, IsSortable = true, IsFacetable =
true)]
    public string Category { get; set; }

    [JsonIgnore]
    public bool? SmokingAllowed => (Rooms != null) ? Array.Exists(Rooms,
element => element.SmokingAllowed == true) : (bool?)null;

    [SearchableField]
    public Address Address { get; set; }

    public Room[] Rooms { get; set; }
```

Choosing a field class

When defining fields, you can use the base SearchField class, or you can use derivative helper models that serve as "templates", with pre-configured properties.

Exactly one field in your index must serve as the document key (`IsKey = true`). It must be a string, and it must uniquely identify each document. It's also required to have `IsHidden = true`, which means it can't be visible in search results.

Field type	Description and usage
SearchField	Base class, with most properties set to null, excepting `Name` which is required, and `AnalyzerName` which defaults to standard Lucene.
SimpleField	Helper model. Can be any data type, is always non-searchable (it's ignored for full text search queries), and is retrievable (it's not hidden). Other attributes are off by default, but can be enabled. You might use a `SimpleField` for document IDs or fields used only in filters, facets, or scoring profiles. If so, be sure to apply any attributes that are necessary for the scenario, such as `IsKey = true` for a document ID. For more information, see SimpleFieldAttribute.cs in source code.
SearchableField	Helper model. Must be a string, and is always searchable and retrievable. Other attributes are off by default, but can be enabled. Because this field type is searchable, it supports synonyms and the full complement of analyzer properties. For more information, see the SearchableFieldAttribute.cs in source code.

Whether you use the basic `SearchField` API or either one of the helper models, you must explicitly enable filter, facet, and sort attributes. For example, IsFilterable, IsSortable, and IsFacetable must be explicitly attributed, as shown in the sample above.

Adding field attributes

Notice how each field is decorated with attributes such as `IsFilterable`, `IsSortable`, `IsKey`, and `AnalyzerName`. These attributes map directly to the corresponding field attributes in an Azure Cognitive Search index. The `FieldBuilder` class uses these properties to construct field definitions for the index.

Field type mapping

The .NET types of the properties map to their equivalent field types in the index definition. For example, the `Category` string property maps to the `category` field, which

is of type `Edm.String`. There are similar type mappings between `bool?`, `Edm.Boolean`, `DateTimeOffset?`, and `Edm.DateTimeOffset` and so on.

Did you happen to notice the `SmokingAllowed` property?

C#

```
[JsonIgnore]
public bool? SmokingAllowed => (Rooms != null) ? Array.Exists(Rooms, element
=> element.SmokingAllowed == true) : (bool?)null;
```

The `JsonIgnore` attribute on this property tells the `FieldBuilder` to not serialize it to the index as a field. This is a great way to create client-side calculated properties you can use as helpers in your application. In this case, the `SmokingAllowed` property reflects whether any `Room` in the `Rooms` collection allows smoking. If all are false, it indicates that the entire hotel doesn't allow smoking.

Load an index

The next step in `Main` populates the newly created "hotels" index. This index population is done in the following method: (Some code replaced with "..." for illustration purposes. See the full sample solution for the full data population code.)

C#

```
private static void UploadDocuments(SearchClient searchClient)
{
    IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "1",
                HotelName = "Secret Point Motel",
                ...
                Address = new Address()
                {
                    StreetAddress = "677 5th Ave",
                    ...
                },
                Rooms = new Room[]
                {
                    new Room()
                    {
                        Description = "Budget Room, 1 Queen Bed (Cityside)",
                        ...
                    },
                    new Room()
```

```
                    {
                        Description = "Budget Room, 1 King Bed (Mountain
View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Deluxe Room, 2 Double Beds (City
View)",
                        ...
                    }
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "2",
                HotelName = "Twin Dome Motel",
                ...
                {
                    StreetAddress = "140 University Town Center Dr",
                    ...
                },
                Rooms = new Room[]
                {
                    new Room()
                    {
                        Description = "Suite, 2 Double Beds (Mountain
View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Standard Room, 1 Queen Bed (City
View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Budget Room, 1 King Bed (Waterfront
View)",
                        ...
                    }
                }
            }),
        IndexDocumentsAction.Upload(
            new Hotel()
            {
                HotelId = "3",
                HotelName = "Triple Landscape Hotel",
                ...
                Address = new Address()
                {
                    StreetAddress = "3393 Peachtree Rd",
                    ...
```

```
                },
                Rooms = new Room[]
                {
                    new Room()
                    {
                        Description = "Standard Room, 2 Queen Beds
(Amenities)",
                        ...
                    },
                    new Room ()
                    {
                        Description = "Standard Room, 2 Double Beds
(Waterfront View)",
                        ...
                    },
                    new Room()
                    {
                        Description = "Deluxe Room, 2 Double Beds
(Cityside)",
                        ...
                    }
                }
            }
        };

    try
    {
        IndexDocumentsResult result = searchClient.IndexDocuments(batch);
    }
    catch (Exception)
    {
        // Sometimes when your Search service is under load, indexing will
fail for some of the documents in
        // the batch. Depending on your application, you can take
compensating actions like delaying and
        // retrying. For this simple demo, we just log the failed document
keys and continue.
        Console.WriteLine("Failed to index some of the documents: {0}");
    }

    Console.WriteLine("Waiting for documents to be indexed...\n");
    Thread.Sleep(2000);
```

This method has four parts. The first creates an array of three `Hotel` objects each with three `Room` objects that will serve as our input data to upload to the index. This data is hard-coded for simplicity. In an actual application, data will likely come from an external data source such as an SQL database.

The second part creates an IndexDocumentsBatch containing the documents. You specify the operation you want to apply to the batch at the time you create it, in this

case by calling IndexDocumentsAction.Upload. The batch is then uploaded to the Azure Cognitive Search index by the IndexDocuments method.

> **Note**
>
> In this example, we are just uploading documents. If you wanted to merge changes into existing documents or delete documents, you could create batches by calling `IndexDocumentsAction.Merge`, `IndexDocumentsAction.MergeOrUpload`, or `IndexDocumentsAction.Delete` instead. You can also mix different operations in a single batch by calling `IndexBatch.New`, which takes a collection of `IndexDocumentsAction` objects, each of which tells Azure Cognitive Search to perform a particular operation on a document. You can create each `IndexDocumentsAction` with its own operation by calling the corresponding method such as `IndexDocumentsAction.Merge`, `IndexAction.Upload`, and so on.

The third part of this method is a catch block that handles an important error case for indexing. If your search service fails to index some of the documents in the batch, a `RequestFailedException` is thrown. An exception can happen if you're indexing documents while your service is under heavy load. **We strongly recommend explicitly handling this case in your code.** You can delay and then retry indexing the documents that failed, or you can log and continue like the sample does, or you can do something else depending on your application's data consistency requirements. An alternative is to use SearchIndexingBufferedSender for intelligent batching, automatic flushing, and retries for failed indexing actions. See this example for more context.

Finally, the `UploadDocuments` method delays for two seconds. Indexing happens asynchronously in your search service, so the sample application needs to wait a short time to ensure that the documents are available for searching. Delays like this are typically only necessary in demos, tests, and sample applications.

Call UploadDocuments in Main()

The following code snippet sets up an instance of SearchClient using the GetSearchClient method of indexClient. The indexClient uses an admin API key on its requests, which is required for loading or refreshing documents.

An alternate approach is to call `SearchClient` directly, passing in an admin API key on `AzureKeyCredential`.

```
C#
```

```
SearchClient searchClient = indexClient.GetSearchClient(indexName);

Console.WriteLine("{0}", "Uploading documents...\n");
UploadDocuments(searchClient);
```

Run queries

First, set up a `SearchClient` that reads the search endpoint and query API key from **appsettings.json**:

```C#
private static SearchClient CreateSearchClientForQueries(string indexName,
IConfigurationRoot configuration)
{
    string searchServiceEndPoint = configuration["SearchServiceEndPoint"];
    string queryApiKey = configuration["SearchServiceQueryApiKey"];

    SearchClient searchClient = new SearchClient(new
Uri(searchServiceEndPoint), indexName, new AzureKeyCredential(queryApiKey));
    return searchClient;
}
```

Second, define a method that sends a query request.

Each time the method executes a query, it creates a new SearchOptions object. This object is used to specify additional options for the query such as sorting, filtering, paging, and faceting. In this method, we're setting the `Filter`, `Select`, and `OrderBy` property for different queries. For more information about the search query expression syntax, Simple query syntax.

The next step is query execution. Running the search is done using the `SearchClient.Search` method. For each query, pass the search text to use as a string (or `"*"` if there's no search text), plus the search options created earlier. We also specify `Hotel` as the type parameter for `SearchClient.Search`, which tells the SDK to deserialize documents in the search results into objects of type `Hotel`.

```C#
private static void RunQueries(SearchClient searchClient)
{
    SearchOptions options;
    SearchResults<Hotel> results;

    Console.WriteLine("Query 1: Search for 'motel'. Return only the
```

```
HotelName in results:\n");

    options = new SearchOptions();
    options.Select.Add("HotelName");

    results = searchClient.Search<Hotel>("motel", options);

    WriteDocuments(results);

    Console.Write("Query 2: Apply a filter to find hotels with rooms cheaper
than $100 per night, ");
    Console.WriteLine("returning the HotelId and Description:\n");

    options = new SearchOptions()
    {
        Filter = "Rooms/any(r: r/BaseRate lt 100)"
    };
    options.Select.Add("HotelId");
    options.Select.Add("Description");

    results = searchClient.Search<Hotel>("*", options);

    WriteDocuments(results);

    Console.Write("Query 3: Search the entire index, order by a specific
field (lastRenovationDate) ");
    Console.Write("in descending order, take the top two results, and show
only hotelName and ");
    Console.WriteLine("lastRenovationDate:\n");

    options =
        new SearchOptions()
        {
            Size = 2
        };
    options.OrderBy.Add("LastRenovationDate desc");
    options.Select.Add("HotelName");
    options.Select.Add("LastRenovationDate");

    results = searchClient.Search<Hotel>("*", options);

    WriteDocuments(results);

    Console.WriteLine("Query 4: Search the HotelName field for the term
'hotel':\n");

    options = new SearchOptions();
    options.SearchFields.Add("HotelName");

    //Adding details to select, because "Location" isn't supported yet when
deserializing search result to "Hotel"
    options.Select.Add("HotelId");
    options.Select.Add("HotelName");
    options.Select.Add("Description");
    options.Select.Add("Category");
```

```
    options.Select.Add("Tags");
    options.Select.Add("ParkingIncluded");
    options.Select.Add("LastRenovationDate");
    options.Select.Add("Rating");
    options.Select.Add("Address");
    options.Select.Add("Rooms");

    results = searchClient.Search<Hotel>("hotel", options);

    WriteDocuments(results);
}
```

Third, define a method that writes the response, printing each document to the console:

```C#
private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.GetResults())
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}
```

Call RunQueries in Main()

```C#
SearchClient indexClientForQueries = CreateSearchClientForQueries(indexName,
configuration);

Console.WriteLine("{0}", "Running queries...\n");
RunQueries(indexClientForQueries);
```

Explore query constructs

Let's take a closer look at each of the queries in turn. Here's the code to execute the first query:

```C#
options = new SearchOptions();
options.Select.Add("HotelName");

results = searchClient.Search<Hotel>("motel", options);
```

```
WriteDocuments(results);
```

In this case, we're searching the entire index for the word "motel" in any searchable field and we only want to retrieve the hotel names, as specified by the `Select` option. Here are the results:

Output

```
Name: Secret Point Motel

Name: Twin Dome Motel
```

In the second query, use a filter to select room with a nightly rate of less than $100. Return only the hotel ID and description in the results:

C#

```
options = new SearchOptions()
{
    Filter = "Rooms/any(r: r/BaseRate lt 100)"
};
options.Select.Add("HotelId");
options.Select.Add("Description");

results = searchClient.Search<Hotel>("*", options);
```

The above query uses an OData `$filter` expression, `Rooms/any(r: r/BaseRate lt 100)`, to filter the documents in the index. This uses the any operator to apply the 'BaseRate lt 100' to every item in the Rooms collection. For more information, see OData filter syntax.

In the third query, find the top two hotels that have been most recently renovated, and show the hotel name and last renovation date. Here's the code:

C#

```
options =
    new SearchOptions()
    {
        Size = 2
    };
options.OrderBy.Add("LastRenovationDate desc");
options.Select.Add("HotelName");
options.Select.Add("LastRenovationDate");

results = searchClient.Search<Hotel>("*", options);
```

```
WriteDocuments(results);
```

In the last query, find all hotels names that match the word "hotel":

```C#
options.Select.Add("HotelId");
options.Select.Add("HotelName");
options.Select.Add("Description");
options.Select.Add("Category");
options.Select.Add("Tags");
options.Select.Add("ParkingIncluded");
options.Select.Add("LastRenovationDate");
options.Select.Add("Rating");
options.Select.Add("Address");
options.Select.Add("Rooms");

results = searchClient.Search<Hotel>("hotel", options);

WriteDocuments(results);
```

This section concludes this introduction to the .NET SDK, but don't stop here. The next section suggests other resources for learning more about programming with Azure Cognitive Search.

Next steps

- Browse the API reference documentation for Azure.Search.Documents and REST API
- Browse other code samples based on Azure.Search.Documents in azure-search-dotnet-samples and search-getting-started-dotnet
- Review naming conventions to learn the rules for naming various objects
- Review supported data types

Additional resources

Documentation

Azure Search SDK for .NET - Azure for .NET Developers

Reference for Azure Search SDK for .NET

Quickstart: Create a search index in .NET - Azure Cognitive Search

In this C# quickstart, learn how to create an index, load data, and run queries using the Azure.Search.Documents client library.

Azure Cognitive Search client library for .NET - Azure for .NET Developers

SearchClient Class (Azure.Search.Documents) - Azure for .NET Developers

Azure Cognitive Search client that can be used to query an index and upload, merge, or delete documents.

SearchIndexClient Class (Azure.Search.Documents.Indexes) - Azure for .NET Developers

Azure Cognitive Search client that can be used to manage indexes on a Search service.

Show 2 more

Upgrade to Azure Cognitive Search .NET SDK version 11

Article • 12/27/2022 • 8 minutes to read

If your search solution is built on the **Azure SDK for .NET**, this article will help you migrate your code from earlier versions of **Microsoft.Azure.Search** to version 11, the new **Azure.Search.Documents** client library. Version 11 is a fully redesigned client library, released by the Azure SDK development team (previous versions were produced by the Azure Cognitive Search development team).

All features from version 10 are implemented in version 11. Key differences include:

- One package (**Azure.Search.Documents**) instead of four
- Three clients instead of two: SearchClient, SearchIndexClient, SearchIndexerClient
- Naming differences across a range of APIs and small structural differences that simplify some tasks

The client library's Change Log has an itemized list of updates. You can review a summarized version in this article.

All C# code samples and snippets in the Cognitive Search product documentation have been revised to use the new **Azure.Search.Documents** client library.

Why upgrade?

The benefits of upgrading are summarized as follows:

- New features will be added to **Azure.Search.Documents** only. The previous version, Microsoft.Azure.Search, is now retired. Updates to deprecated libraries are limited to high priority bug fixes only.

- Consistency with other Azure client libraries. **Azure.Search.Documents** takes a dependency on Azure.Core and System.Text.Json, and follows conventional approaches for common tasks such as client connections and authorization.

Microsoft.Azure.Search is officially retired. If you're using an old version, we recommend upgrading to the next higher version, repeating the process in succession until you reach version 11 and **Azure.Search.Documents**. An incremental upgrade strategy makes it easier to find and fix blocking issues. See Previous version docs for guidance.

Package comparison

Version 11 consolidates and simplifies package management so that there are fewer to manage.

Version 10 and earlier	Version 11
Microsoft.Azure.Search Microsoft.Azure.Search.Service Microsoft.Azure.Search.Data Microsoft.Azure.Search.Common	Azure.Search.Documents package

Client comparison

Where applicable, the following table maps the client libraries between the two versions.

Client operations	Microsoft.Azure.Search (v10)	Azure.Search.Documents (v11)
Targets the documents collection of an index (queries and data import)	SearchIndexClient	SearchClient
Targets index-related objects (indexes, analyzers, synonym maps	SearchServiceClient	SearchIndexClient
Targets indexer-related objects (indexers, data sources, skillsets)	SearchServiceClient	SearchIndexerClient (**new**)

> **Caution**
>
> Notice that SearchIndexClient exists in both versions, but targets different operations. In version 10, SearchIndexClient creates indexes and other objects. In version 11, SearchIndexClient works with existing indexes, targeting the documents collection with query and data ingestion APIs. To avoid confusion when updating code, be mindful of the order in which client references are updated. Following the sequence in **Steps to upgrade** should help mitigate any string replacement issues.

Naming and other API differences

Besides the client differences (noted previously and thus omitted here), multiple other APIs have been renamed and in some cases redesigned. Class name differences are

summarized below. This list isn't exhaustive but it does group API changes by task, which can be helpful for revisions on specific code blocks. For an itemized list of API updates, see the change log for `Azure.Search.Documents` on GitHub.

Authentication and encryption

Version 10	Version 11 equivalent
SearchCredentials	AzureKeyCredential
EncryptionKey (Undocumented in API reference. Support for this API transitioned to generally available in v10, but was only available in the preview SDK)	SearchResourceEncryptionKey

Indexes, analyzers, synonym maps

Version 10	Version 11 equivalent
Index	SearchIndex
Field	SearchField
DataType	SearchFieldDataType
ItemError	SearchIndexerError
Analyzer	LexicalAnalyzer (also, `AnalyzerName` to `LexicalAnalyzerName`)
AnalyzeRequest	AnalyzeTextOptions
StandardAnalyzer	LuceneStandardAnalyzer
StandardTokenizer	LuceneStandardTokenizer (also, `StandardTokenizerV2` to `LuceneStandardTokenizerV2`)
TokenInfo	AnalyzedTokenInfo
Tokenizer	LexicalTokenizer (also, `TokenizerName` to `LexicalTokenizerName`)
SynonymMap.Format	None. Remove references to `Format`.

Field definitions are streamlined: SearchableField, SimpleField, ComplexField are new APIs for creating field definitions.

Indexers, datasources, skillsets

Version 10	Version 11 equivalent
Indexer	SearchIndexer
DataSource	SearchIndexerDataSourceConnection
Skill	SearchIndexerSkill
Skillset	SearchIndexerSkillset
DataSourceType	SearchIndexerDataSourceType

Data import

Version 10	Version 11 equivalent
IndexAction	IndexDocumentsAction
IndexBatch	IndexDocumentsBatch
IndexBatchException.FindFailedActionsToRetry()	SearchIndexingBufferedSender

Query requests and responses

Version 10	Version 11 equivalent
DocumentsOperationsExtensions.SearchAsync	SearchClient.SearchAsync
DocumentSearchResult	SearchResult or SearchResults, depending on whether the result is a single document or multiple.
DocumentSuggestResult	SuggestResults
SearchParameters	SearchOptions
SuggestParameters	SuggestOptions
SearchParameters.Filter	SearchFilter (a new class for constructing OData filter expressions)

JSON serialization

By default, the Azure SDK uses System.Text.Json for JSON serialization, relying on the capabilities of those APIs to handle text transformations previously implemented

through a native SerializePropertyNamesAsCamelCaseAttribute class, which has no counterpart in the new library.

To serialize property names into camelCase, you can use the JsonPropertyNameAttribute (similar to this example).

Alternatively, you can set a JsonNamingPolicy provided in JsonSerializerOptions. The following System.Text.Json code example, taken from the Microsoft.Azure.Core.Spatial readme demonstrates the use of camelCase without having to attribute every property:

```C#
// Get the Azure Cognitive Search endpoint and read-only API key.
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
AzureKeyCredential credential = new
AzureKeyCredential(Environment.GetEnvironmentVariable("SEARCH_API_KEY"));

// Create serializer options with our converter to deserialize geographic
points.
JsonSerializerOptions serializerOptions = new JsonSerializerOptions
{
    Converters =
    {
        new MicrosoftSpatialGeoJsonConverter()
    },
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
};

SearchClientOptions clientOptions = new SearchClientOptions
{
    Serializer = new JsonObjectSerializer(serializerOptions)
};

SearchClient client = new SearchClient(endpoint, "mountains", credential,
clientOptions);
Response<SearchResults<Mountain>> results = client.Search<Mountain>
("Rainier");
```

If you're using Newtonsoft.Json for JSON serialization, you can pass in global naming policies using similar attributes, or by using properties on JsonSerializerSettings. For an example equivalent to the one above, see the Deserializing documents example in the Newtonsoft.Json readme.

Inside v11

Each version of an Azure Cognitive Search client library targets a corresponding version of the REST API. The REST API is considered foundational to the service, with individual SDKs wrapping a version of the REST API. As a .NET developer, it can be helpful to review the more verbose REST API documentation for more in depth coverage of specific objects or operations. Version 11 targets the 2020-06-30 search service.

Version 11.0 fully supports the following objects and operations:

- Index creation and management
- Synonym map creation and management
- Indexer creation and management
- Indexer data source creation and management
- Skillset creation and management
- All query types and syntax

Version 11.1 additions (change log details):

- FieldBuilder (added in 11.1)
- Serializer property (added in 11.1) to support custom serialization

Version 11.2 additions (change log details):

- EncryptionKey property added indexers, data sources, and skillsets

- IndexingParameters.IndexingParametersConfiguration property support

- Geospatial types are natively supported in FieldBuilder. SearchFilter can encode geometric types from Microsoft.Spatial without an explicit assembly dependency.

  You can also continue to explicitly declare a dependency on Microsoft.Spatial. Examples of this technique are available for System.Text.Json and Newtonsoft.Json.

Version 11.3 additions (change log details):

- KnowledgeStore
- Added support for Azure.Core.GeoJson types in SearchDocument, SearchFilter and FieldBuilder.
- Added EventSource based logging. Event source name is Azure-Search-Documents. Current set of events are focused on tuning batch sizes for SearchIndexingBufferedSender.
- Added CustomEntityLookupSkill and DocumentExtractionSkill. Added DefaultCountryHint in LanguageDetectionSkill.

Before upgrading

- Quickstarts, tutorials, and C# samples have been updated to use the Azure.Search.Documents package. We recommend reviewing the samples and walkthroughs to learn about the new APIs before embarking on a migration exercise.

- How to use Azure.Search.Documents introduces the most commonly used APIs. Even knowledgeable users of Cognitive Search might want to review this introduction to the new library as a precursor to migration.

Steps to upgrade

The following steps get you started on a code migration by walking through the first set of required tasks, especially with regard to client references.

1. Install the Azure.Search.Documents package by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

2. Replace using directives for Microsoft.Azure.Search with the following:

   C#

   ```
   using Azure;
   using Azure.Search.Documents;
   using Azure.Search.Documents.Indexes;
   using Azure.Search.Documents.Indexes.Models;
   using Azure.Search.Documents.Models;
   ```

3. For classes that require JSON serialization, replace `using Newtonsoft.Json` with `using System.Text.Json.Serialization`.

4. Revise client authentication code. In previous versions, you would use properties on the client object to set the API key (for example, the SearchServiceClient.Credentials property). In the current version, use the AzureKeyCredential class to pass the key as a credential, so that if needed, you can update the API key without creating new client objects.

   Client properties have been streamlined to just `Endpoint`, `ServiceName`, and `IndexName` (where appropriate). The following example uses the system Uri class to provide the endpoint and the Environment class to read in the key value:

   C#

```
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
AzureKeyCredential credential = new AzureKeyCredential(
   Environment.GetEnvironmentVariable("SEARCH_API_KEY"));
SearchIndexClient indexClient = new SearchIndexClient(endpoint,
credential);
```

5. Add new client references for indexer-related objects. If you're using indexers, datasources, or skillsets, change the client references to SearchIndexerClient. This client is new in version 11 and has no antecedent.

6. Revise collections and lists. In the new SDK, all lists are read-only to avoid downstream issues if the list happens to contain null values. The code change is to add items to a list. For example, instead of assigning strings to a Select property, you would add them as follows:

```C#
var options = new SearchOptions
 {
    SearchMode = SearchMode.All,
    IncludeTotalCount = true
 };

 // Select fields to return in results.
 options.Select.Add("HotelName");
 options.Select.Add("Description");
 options.Select.Add("Tags");
 options.Select.Add("Rooms");
 options.Select.Add("Rating");
 options.Select.Add("LastRenovationDate");
```

   Select, Facets, SearchFields, SourceFields, ScoringParameters, and OrderBy are all lists that now need to be reconstructed.

7. Update client references for queries and data import. Instances of SearchIndexClient should be changed to SearchClient. To avoid name confusion, make sure you catch all instances before proceeding to the next step.

8. Update client references for index, synonym map, and analyzer objects. Instances of SearchServiceClient should be changed to SearchIndexClient.

9. For the remainder of your code, update classes, methods, and properties to use the APIs of the new library. The naming differences section is a place to start but you can also review the change log.

If you have trouble finding equivalent APIs, we suggest logging an issue on https://github.com/MicrosoftDocs/azure-docs/issues so that we can improve the documentation or investigate the problem.

10. Rebuild the solution. After fixing any build errors or warnings, you can make additional changes to your application to take advantage of new functionality.

Breaking changes

Given the sweeping changes to libraries and APIs, an upgrade to version 11 is non-trivial and constitutes a breaking change in the sense that your code will no longer be backward compatible with version 10 and earlier. For a thorough review of the differences, see the change log for `Azure.Search.Documents`.

In terms of service version updates, where code changes in version 11 relate to existing functionality (and not just a refactoring of the APIs), you'll find the following behavior changes:

- BM25 ranking algorithm replaces the previous ranking algorithm with newer technology. New services will use this algorithm automatically. For existing services, you must set parameters to use the new algorithm.
- Ordered results for null values have changed in this version, with null values appearing first if the sort is `asc` and last if the sort is `desc`. If you wrote code to handle how null values are sorted, you should review and potentially remove that code if it's no longer necessary.

Due to these behavior changes, it's likely that you'll see slight variations in ranked results.

Next steps

- How to use Azure.Search.Documents in a C# .NET Application
- Tutorial: Add search to web apps
- Azure.Search.Documents package
- Samples on GitHub
- Azure.Search.Document API reference

Additional resources

Documentation

Azure Cognitive Search client library for .NET - Azure for .NET Developers

SearchIndexClient Class (Azure.Search.Documents.Indexes) - Azure for .NET Developers

Azure Cognitive Search client that can be used to manage indexes on a Search service.

SearchClient Class (Azure.Search.Documents) - Azure for .NET Developers

Azure Cognitive Search client that can be used to query an index and upload, merge, or delete documents.

SearchOptions Class (Azure.Search.Documents) - Azure for .NET Developers

Parameters for filtering, sorting, faceting, paging, and other search query behaviors.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

.NET samples - Azure Cognitive Search

Find Azure Cognitive Search demo C# code samples that use the .NET client libraries.

Show 3 more

Upgrading versions of the Azure Search .NET Management SDK

Article • 10/12/2022 • 3 minutes to read

This article explains how to migrate to successive versions of the Azure Search .NET Management SDK, used to provision or deprovision search services, adjust capacity, and manage API keys.

Management SDKs target a specific version of the Management REST API. For more information about concepts and operations, see Search Management (REST).

Versions

Microsoft.Azure.Management.Search is now deprecated. We recommend Azure.ResourceManager.Search instead.

SDK version	Corresponding REST API version	Feature addition or behavior change
1.0	api-version=2020-08-01	This is a new package from the Azure SDK team that implements approaches and standards that are common to resource management in Azure. There's no migration path. If you've used the previous client library for service administration in Azure Cognitive Search, you should redesign your solution to use the new `Azure.ResourceManager.Search` package. See the readme for links and next steps.
3.0	api-version=2020-30-20	Adds endpoint security (IP firewalls and integration with Azure Private Link)
2.0	api-version=2019-10-01	Usability improvements. Breaking change on List Query Keys (GET is discontinued).
1.0	api-version=2015-08-19	First version

How to upgrade

1. Review the client library changelist for insight into the scope of changes.

2. In your application code, delete the reference to `Microsoft.Azure.Management.Search` and its dependencies.

3. Add a reference for `Azure.ResourceManager.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

4. Once NuGet has downloaded the new packages and their dependencies, replace the API calls.

Upgrade to 3.0

Version 3.0 adds private endpoint protection by restricting access to IP ranges, and by optionally integrating with Azure Private Link for search services that shouldn't be visible on the public internet.

New APIs

API	Category	Details
NetworkRuleSet	IP firewall	Restrict access to a service endpoint to a list of allowed IP addresses. See Configure IP firewall for concepts and portal instructions.
Shared Private Link Resource	Private Link	Create a shared private link resource to be used by a search service.
Private Endpoint Connections	Private Link	Establish and manage connections to a search service through private endpoint. See Create a private endpoint for concepts and portal instructions.
Private Link Resources	Private Link	For a search service that has a private endpoint connection, get a list of all services used in the same virtual network. If your search solution includes indexers that pull from Azure data sources (such as Azure Storage, Azure Cosmos DB, Azure SQL), or uses Cognitive Services or Key Vault, then all of those resources should have endpoints in the virtual network, and this API should return a list.
PublicNetworkAccess	Private Link	This is a property on Create or Update Service requests. When disabled, private link is the only access modality.

Breaking changes

You can no longer use GET on a List Query Keys request. In previous releases you could use either GET or POST, in this release and in all releases moving forward, only POST is supported.

Upgrade to 2.0

Version 2 of the Azure Search .NET Management SDK is a minor upgrade, so changing your code should require only minimal effort. The changes to the SDK are strictly client-side changes to improve the usability of the SDK itself. These changes include the following:

- `Services.CreateOrUpdate` and its asynchronous versions now automatically poll the provisioning `SearchService` and don't return until service provisioning is complete. This saves you from having to write such polling code yourself.
- If you still want to poll service provisioning manually, you can use the new `Services.BeginCreateOrUpdate` method or one of its asynchronous versions.
- New methods `Services.Update` and its asynchronous versions have been added to the SDK. These methods use HTTP PATCH to support incremental updating of a service. For example, you can now scale a service by passing a `SearchService` instance to these methods that contains only the desired `partitionCount` and `replicaCount` properties. The old way of calling `Services.Get`, modifying the returned `SearchService`, and passing it to `Services.CreateOrUpdate` is still supported, but is no longer necessary.

Next steps

If you encounter problems, the best forum for posting questions is Stack Overflow. If you find a bug, you can file an issue in the Azure .NET SDK GitHub repository. Make sure to label your issue title with "[search]".

Upgrade to the latest REST API in Azure Cognitive Search

Article • 10/12/2022 • 7 minutes to read

If you're using an earlier version of the **Search REST API**, this article will help you upgrade your application to the newest generally available API version, **2020-06-30**.

Version 2020-06-30 includes an important new feature (knowledge store), and introduces several minor behavior changes. As such, this version is mostly backward compatible so code changes should be minimal if you're upgrading from the previous version (2019-05-06).

> **Note**
>
> A search service supports a range of REST API versions, including earlier ones. You can continue to use those API versions, but we recommend migrating your code to the newest version so that you can access new capabilities. Over time, the most outdated versions of the REST API will be deprecated and **no longer supported**.

How to upgrade

When upgrading to a new version, you probably won't have to make many changes to your code, other than to change the version number. The only situations in which you may need to change code are when:

- Your code fails when unrecognized properties are returned in an API response. By default your application should ignore properties that it doesn't understand.
- Your code persists API requests and tries to resend them to the new API version. For example, this might happen if your application persists continuation tokens returned from the Search API (for more information, look for `@search.nextPageParameters` in the Search API Reference).
- Your code references an API version that predates 2019-05-06 and is subject to one or more of the breaking changes in that release. The section Upgrade to 2019-05-06 provides more detail.

If any of these situations apply to you, then you may need to change your code accordingly. Otherwise, no changes should be necessary, although you might want to start using features added in the new version.

Upgrade to 2020-06-30

Version 2020-06-30 is the new generally available release of the REST API. There's one breaking change and several behavioral differences.

Features are now generally available in this API version include:

- Knowledge store, persistent storage of enriched content created through skillsets, created for downstream analysis and processing through other applications. With this capability, an indexer-driven AI enrichment pipeline can populate a knowledge store in addition to a search index. If you used the preview version of this feature, it's equivalent to the generally available version. The only code change required is modifying the api-version.

Breaking change

Existing code written against earlier API versions will break on api-version=2020-06-30 and later if code contains the following functionality:

- Any Edm.Date literals (a date composed of year-month-day, such as `2020-12-12`) in filter expressions must follow the Edm.DateTimeOffset format: `2020-12-12T00:00:00Z`. This change was necessary to handle erroneous or unexpected query results due to timezone differences.

Behavior changes

- BM25 ranking algorithm replaces the previous ranking algorithm with newer technology. New services will use this algorithm automatically. For existing services, you must set parameters to use the new algorithm.

- Ordered results for null values have changed in this version, with null values appearing first if the sort is `asc` and last if the sort is `desc`. If you wrote code to handle how null values are sorted, be aware of this change.

Upgrade to 2019-05-06

Version 2019-05-06 is the previous generally available release of the REST API. Features that became generally available in this API version include:

- Autocomplete is a typeahead feature that completes a partially specified term input.

- Complex types provides native support for structured object data in search index.
- JsonLines parsing modes, part of Azure Blob indexing, creates one search document per JSON entity that is separated by a newline.
- AI enrichment provides indexing that uses the AI enrichment engines of Cognitive Services.

Breaking changes

Existing code written against earlier API versions will break on api-version=2019-05-06 and later if code contains the following functionality:

Indexer for Azure Cosmos DB - datasource is now `"type": "cosmosdb"`

If you're using an Azure Cosmos DB indexer, you must change `"type": "documentdb"` to `"type": "cosmosdb"`.

Indexer execution result errors no longer have status

The error structure for indexer execution previously had a `status` element. This element was removed because it wasn't providing useful information.

Indexer data source API no longer returns connection strings

From API versions 2019-05-06 and 2019-05-06-Preview onwards, the data source API no longer returns connection strings in the response of any REST operation. In previous API versions, for data sources created using POST, Azure Cognitive Search returned **201** followed by the OData response, which contained the connection string in plain text.

Named Entity Recognition cognitive skill is now discontinued

If you called the Name Entity Recognition skill in your code, the call will fail. Replacement functionality is Entity Recognition Skill (V3). Follow the recommendations in Deprecated cognitive search skills to migrate to a supported skill.

Upgrading complex types

API version 2019-05-06 added formal support for complex types. If your code implemented previous recommendations for complex type equivalency in 2017-11-11-

Preview or 2016-09-01-Preview, there are some new and changed limits starting in version 2019-05-06 of which you need to be aware:

- The limits on the depth of subfields and the number of complex collections per index have been lowered. If you created indexes that exceed these limits using the preview api-versions, any attempt to update or recreate them using API version 2019-05-06 will fail. If you find yourself in this situation, you'll need to redesign your schema to fit within the new limits and then rebuild your index.
- There's a new limit starting in api-version 2019-05-06 on the number of elements of complex collections per document. If you created indexes with documents that exceed these limits using the preview api-versions, any attempt to reindex that data using api-version 2019-05-06 will fail. If you find yourself in this situation, you'll need to reduce the number of complex collection elements per document before reindexing your data.

For more information, see Service limits for Azure Cognitive Search.

How to upgrade an old complex type structure

If your code is using complex types with one of the older preview API versions, you may be using an index definition format that looks like this:

JSON

```
{
  "name": "hotels",
  "fields": [
    { "name": "HotelId", "type": "Edm.String", "key": true, "filterable":
true },
    { "name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": true, "facetable": false },
    { "name": "Description", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"en.microsoft" },
    { "name": "Description_fr", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"fr.microsoft" },
    { "name": "Category", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
    { "name": "Tags", "type": "Collection(Edm.String)", "searchable": true,
"filterable": true, "sortable": false, "facetable": true, "analyzer":
"tagsAnalyzer" },
    { "name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true,
"sortable": true, "facetable": true },
    { "name": "LastRenovationDate", "type": "Edm.DateTimeOffset",
"filterable": true, "sortable": true, "facetable": true },
    { "name": "Rating", "type": "Edm.Double", "filterable": true,
```

```
"sortable": true, "facetable": true },
    { "name": "Address", "type": "Edm.ComplexType" },
    { "name": "Address/StreetAddress", "type": "Edm.String", "filterable":
false, "sortable": false, "facetable": false, "searchable": true },
    { "name": "Address/City", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
    { "name": "Address/StateProvince", "type": "Edm.String", "searchable":
true, "filterable": true, "sortable": true, "facetable": true },
    { "name": "Address/PostalCode", "type": "Edm.String", "searchable":
true, "filterable": true, "sortable": true, "facetable": true },
    { "name": "Address/Country", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
    { "name": "Location", "type": "Edm.GeographyPoint", "filterable": true,
"sortable": true },
    { "name": "Rooms", "type": "Collection(Edm.ComplexType)" },
    { "name": "Rooms/Description", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false, "analyzer":
"en.lucene" },
    { "name": "Rooms/Description_fr", "type": "Edm.String", "searchable":
true, "filterable": false, "sortable": false, "facetable": false,
"analyzer": "fr.lucene" },
    { "name": "Rooms/Type", "type": "Edm.String", "searchable": true },
    { "name": "Rooms/BaseRate", "type": "Edm.Double", "filterable": true,
"facetable": true },
    { "name": "Rooms/BedOptions", "type": "Edm.String", "searchable": true
},
    { "name": "Rooms/SleepsCount", "type": "Edm.Int32", "filterable": true,
"facetable": true },
    { "name": "Rooms/SmokingAllowed", "type": "Edm.Boolean", "filterable":
true, "facetable": true },
    { "name": "Rooms/Tags", "type": "Collection(Edm.String)", "searchable":
true, "filterable": true, "facetable": true, "analyzer": "tagsAnalyzer" }
  ]
}
```

A newer tree-like format for defining index fields was introduced in API version 2017-11-11-Preview. In the new format, each complex field has a fields collection where its subfields are defined. In API version 2019-05-06, this new format is used exclusively and attempting to create or update an index using the old format will fail. If you have indexes created using the old format, you'll need to use API version 2017-11-11-Preview to update them to the new format before they can be managed using API version 2019-05-06.

You can update "flat" indexes to the new format with the following steps using API version 2017-11-11-Preview:

1. Perform a GET request to retrieve your index. If it’s already in the new format, you’re done.

2. Translate the index from the “flat” format to the new format. You’ll have to write code for this task since there's no sample code available at the time of this writing.

3. Perform a PUT request to update the index to the new format. Avoid changing any other details of the index, such as the searchability/filterability of fields, because changes that affect the physical expression of existing index isn't allowed by the Update Index API.

> **Note**
>
> It is not possible to manage indexes created with the old "flat" format from the Azure portal. Please upgrade your indexes from the “flat” representation to the “tree” representation at your earliest convenience.

Next steps

Review the Search REST API reference documentation. If you encounter problems, ask us for help on Stack Overflow or contact support.

Search service REST API Reference

Create an index in Azure Cognitive Search

Article • 05/05/2023

In Azure Cognitive Search, query requests target the searchable text in a **search index**.

In this article, learn the steps for defining and publishing a search index. Creating an index establishes the physical data structure (folders and files) on your search service. Once the index definition exists, **loading the index** follows as a separate task.

Prerequisites

- Write permissions on the search service. Permission can be granted through an admin API key on the request. Alternatively, if you're using role-based access control, you can issue your request as a member of the Search Contributor role.
- An external data source that provides the content to be indexed. You should refer to the data source to understand the schema requirements of your search index. Index creation is largely a schema definition exercise. Before creating one, you should have:
  - A clear idea of which source fields you want to make searchable, retrievable, filterable, facetable, and sortable in the search index (see the schema checklist for guidance).
  - A unique field in source data that can be used as the document key (or ID) in the index.
- A stable index location. Moving an existing index to a different search service is not supported out-of-the-box. Revisit application requirements and make sure that your existing search service, its capacity and location, are sufficient for your needs.
- Finally, all service tiers have index limits on the number of objects that you can create. For example, if you are experimenting on the Free tier, you can only have 3 indexes at any given time. Within the index itself, there are limits on the number of complex fields and collections.

Document keys

A search index has one required field: a document key. A document key is the unique identifier of a search document. In Azure Cognitive Search, it must be a string, and it must originate from unique values in the data source that's providing the content to be indexed. A search service does not generate key values, but in some scenarios (such as the Azure Table indexer) it will synthesize existing values to create a unique key for the documents being indexed.

During incremental indexing, where just new and updated content is indexed, incoming documents with new keys are added, while incoming documents with existing keys are either merged or overwritten, depending on whether index fields are null or populated.

Schema checklist

Use this checklist to assist the design decisions for your search index.

1. Review naming conventions so that index and field names conform to the naming rules.

2. Review supported data types. The data type will impact how the field is used. For example, numeric content is filterable but not full text searchable. The most common data type is `Edm.String` for searchable text, which is tokenized and queried using the full text search engine.

3. Identify a document key. A document key is an index requirement. It's a single string field and it will be populated from a source data field that contains unique values. For example, if you're indexing from Blob Storage, the metadata storage path is often used as the document key because it uniquely identifies each blob in the container.

4. Identify the fields in your data source that will contribute searchable content in the index. Searchable content includes short or long strings that are queried using the full text search engine. If the content is verbose (small phrases or bigger chunks), experiment with different analyzers to see how the text is tokenized.

   Field attribute assignments will determine both search behaviors and the physical representation of your index on the search service. Determining how fields should be specified is an iterative process for many customers. To speed up iterations, start with sample data so that you can drop and rebuild easily.

5. Identify which source fields can be used as filters. Numeric content and short text fields, particularly those with repeating values, are good choices. When working with filters, remember:

- Filterable fields can optionally be used in faceted navigation.
- Filterable fields are returned in arbitrary order, so consider making them sortable as well.

6. Determine whether you'll use the default analyzer (`"analyzer": null`) or a different analyzer. Analyzers are used to tokenize text fields during indexing and query execution. If strings are descriptive and semantically rich, or if you have translated strings, consider overriding the default with a language analyzer.

> **Note**
>
> Full text search is conducted over terms that are tokenized during indexing. If your queries fail to return the results you expect, **test for tokenization** to verify the string actually exists. You can try different analyzers on strings to see how tokens are produced for various analyzers.

Create an index

When you're ready to create the index, use a search client that can send the request. You can use the Azure portal or REST APIs for early development and proof-of-concept testing.

During development, plan on frequent rebuilds. Because physical structures are created in the service, dropping and re-creating indexes is necessary for many modifications. You might consider working with a subset of your data to make rebuilds go faster.

Azure portal

Index design through the portal enforces requirements and schema rules for specific data types, such as disallowing full text search capabilities on numeric fields.

1. Sign in to Azure portal.
2. In the search service Overview page, choose either option for creating a search index:
   - **Add index**, an embedded editor for specifying an index schema
   - **Import data wizard**

The wizard is an end-to-end workflow that creates an indexer, a data source, and a finished index. It also loads the data. If this is more than what you want, use **Add index** instead.

The following screenshot highlights where **Add index** and **Import data** appear on the command bar. After an index is created, you can find it again in the **Indexes** tab.

> **Tip**
>
> After creating an index in the portal, you can copy the JSON representation and add it to your application code.

Set `corsOptions` for cross-origin queries

Index schemas include a section for setting `corsOptions`. Client-side JavaScript cannot call any APIs by default since the browser will prevent all cross-origin requests. To allow cross-origin queries to your index, enable CORS (Cross-Origin Resource Sharing) by setting the **corsOptions** attribute. For security reasons, only query APIs support CORS.

JSON

```
"corsOptions": {
  "allowedOrigins": [
    "*"
  ],
  "maxAgeInSeconds": 300
```

The following properties can be set for CORS:

- **allowedOrigins** (required): This is a list of origins that will be granted access to your index. This means that any JavaScript code served from those origins will be allowed to query your index (assuming it provides the correct api-key). Each origin

is typically of the form `protocol://<fully-qualified-domain-name>:<port>` although `<port>` is often omitted. See Cross-origin resource sharing (Wikipedia) for more details.

If you want to allow access to all origins, include `*` as a single item in the **allowedOrigins** array. *This is not a recommended practice for production search services* but it is often useful for development and debugging.

- **maxAgeInSeconds** (optional): Browsers use this value to determine the duration (in seconds) to cache CORS preflight responses. This must be a non-negative integer. The larger this value is, the better performance will be, but the longer it will take for CORS policy changes to take effect. If it is not set, a default duration of 5 minutes will be used.

Allowed updates on existing indexes

Create Index creates the physical data structures (files and inverted indices) on your search service. Once the index is created, your ability to effect changes using **Update Index** is contingent upon whether your modifications invalidate those physical structures. Most field attributes can't be changed once the field is created in your index.

Alternatively, you can create an index alias that serves as a stable reference in your application code. Instead of updating your code, you can update an index alias to point to newer index versions.

To minimize churn in the design process, the following table describes which elements are fixed and flexible in the schema. Changing a fixed element requires an index rebuild, whereas flexible elements can be changed at any time without impacting the physical implementation.

Element	Can be updated?
Name	No
Key	No
Field names and types	No
Field attributes (searchable, filterable, facetable, sortable)	No

Element	Can be updated?
Field attribute (retrievable)	Yes
Analyzer	You can add and modify custom analyzers in the index. Regarding analyzer assignments on string fields, you can only modify "searchAnalyzer". All other assignments and modifications require a rebuild.
Scoring profiles	Yes
Suggesters	No
cross-origin remote scripting (CORS)	Yes
Encryption	Yes

Next steps

Use the following links to become familiar with loading an index with data, or extending an index with a synonyms map.

- Data import overview
- Add, Update or Delete Documents (REST)
- Synonym maps

Load data into a search index in Azure Cognitive Search

Article • 10/21/2022 • 4 minutes to read

This article explains how to import, refresh, and manage content in a predefined search index. In Azure Cognitive Search, a search index is created first, with data import following as a second step. The exception is Import Data wizard, which creates and loads an index in one workflow.

A search service imports and indexes text in JSON, used in full text search or knowledge mining scenarios. Text content is obtainable from alphanumeric fields in the external data source, metadata that's useful in search scenarios, or enriched content created by a skillset (skills can extract or infer textual descriptions from images and unstructured content).

Once data is indexed, the physical data structures of the index are locked in. For guidance on what can and can't be changed, see Drop and rebuild an index.

Indexing isn't a background process. A search service will balance indexing and query workloads, but if query latency is too high, you can either add capacity or identify periods of low query activity for loading an index.

Load documents

A search service accepts JSON documents that conform to the index schema.

You can prepare these documents yourself, but if content resides in a supported data source, running an indexer or the Import data wizard can automate document retrieval, JSON serialization, and indexing.

Azure portal

Using Azure portal, the sole means for loading an index is an indexer or running the Import Data wizard. The wizard creates objects. If you want to load an existing index, you'll need to use an alternative approach.

1. Sign in to the Azure portal with your Azure account.
2. Find your search service and on the Overview page, select **Import data** on the command bar to create and populate a search index. You can follow this

link to review the workflow: Quickstart: Create an Azure Cognitive Search index in the Azure portal.

3. Alternatively, you can reset and run an indexer, which is useful if you're adding fields incrementally. Reset forces the indexer to start over, picking up all fields from all source documents.

Delete orphan documents

Azure Cognitive Search supports document-level operations so that you can look up, update, and delete a specific document in isolation. The following example shows how to delete a document. In a search service, documents are unrelated so deleting one will have no impact on the rest of the index.

1. Identify which field is the document key. In the portal, you can view the fields of each index. Document keys are string fields and are denoted with a key icon to make them easier to spot.

2. Check the values of the document key field: `search=*&$select=HotelId`. A simple string is straightforward, but if the index uses a base-64 encoded field, or if search documents were generated from a `parsingMode` setting, you might be working with values that you aren't familiar with.

3. Look up the document to verify the value of the document ID and to review its content before deleting it. Specify the key or document ID in the request. The following examples illustrate a simple string for the Hotels sample index and a base-64 encoded string for the metadata_storage_path key of the cog-search-demo index.

HTTP

```
GET https://[service name].search.windows.net/indexes/hotel-sample-index/docs/1111?api-version=2020-06-30
```

HTTP

```
GET https://[service name].search.windows.net/indexes/cog-search-demo/docs/aHR0cHM6Ly9oZWlkaWJsb2JzdG9yYWdlMi5ibG9iLmNvcmUud2luZG93cy5uZXQvY29nLXNlYXJjaC1kZW1vL2d1dGhyaWUuanBn0?api-version=2020-06-30
```

4. Delete the document to remove it from the search index.

```HTTP
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/index?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]
{
  "value": [
    {
      "@search.action": "delete",
      "id": "1111"
    }
  ]
}
```

See also

- Search indexes overview
- Data import overview
- Import data wizard overview
- Indexers overview

Drop and rebuild an index in Azure Cognitive Search

Article • 02/09/2023 • 5 minutes to read

This article explains how to drop and rebuild an Azure Cognitive Search index. It explains the circumstances under which rebuilds are required, and provides recommendations for mitigating the impact of rebuilds on ongoing query requests. If you have to rebuild frequently, we recommend using index aliases to make it easier to swap which index your application is pointing to.

During active development, it's common to drop and rebuild indexes when you're iterating over index design. Most developers work with a small representative sample of their data to facilitate this process.

Modifications requiring a rebuild

The following table lists the modifications that require an index rebuild.

Action	Description
Delete a field	To physically remove all traces of a field, you have to rebuild the index. When an immediate rebuild isn't practical, you can modify application code to disable access to the "deleted" field or use the $select query parameter to choose which fields are represented in the result set. Physically, the field definition and contents remain in the index until the next rebuild, when you apply a schema that omits the field in question.
Change a field definition	Revising a field name, data type, or specific index attributes (searchable, filterable, sortable, facetable) requires a full rebuild.
Assign an analyzer to a field	Analyzers are defined in an index and then assigned to fields. You can add a new analyzer definition to an index at any time, but you can only *assign* an analyzer when the field is created. This is true for both the **analyzer** and **indexAnalyzer** properties. The **searchAnalyzer** property is an exception (you can assign this property to an existing field).
Update or delete an analyzer definition in an index	You can't delete or change an existing analyzer configuration (analyzer, tokenizer, token filter, or char filter) in the index unless you rebuild the entire index.

Action	Description
Add a field to a suggester	If a field already exists and you want to add it to a Suggesters construct, you must rebuild the index.
Switch tiers	In-place upgrades aren't supported. If you require more capacity, you must create a new service and rebuild your indexes from scratch. To help automate this process, you can use the **index-backup-restore** sample code in this Azure Cognitive Search .NET sample repo. This app will back up your index to a series of JSON files, and then recreate the index in a search service you specify.

Modifications with no rebuild requirement

Many other modifications can be made without impacting existing physical structures. Specifically, the following changes don't require an index rebuild. For these changes, you can update an existing index definition with your changes.

- Add a new field
- Set the **retrievable** attribute on an existing field
- Update **searchAnalyzer** on a field having an existing **indexAnalyzer**
- Add a new analyzer definition in an index (which can be applied to new fields)
- Add, update, or delete scoring profiles
- Add, update, or delete CORS settings
- Add, update, or delete synonymMaps
- Add, update, or delete semantic configurations

When you add a new field, existing indexed documents are given a null value for the new field. On a future data refresh, values from external source data replace the nulls added by Azure Cognitive Search. For more information on updating index content, see Add, Update or Delete Documents.

How to rebuild an index

During development, the index schema changes frequently. You can plan for it by creating indexes that can be deleted, recreated, and reloaded quickly with a small representative data set.

For applications already in production, we recommend creating a new index that runs side by side an existing index to avoid query downtime. Your application code provides redirection to the new index.

1. Determine whether a rebuild is required. If you're just adding fields, or changing some part of the index that is unrelated to fields, you might be able to simply update the definition without deleting, recreating, and fully reloading it.

2. Get an index definition in case you need it for future reference.

3. Drop the existing index, assuming you aren't running new and old indexes side by side.

   Any queries targeting that index are immediately dropped. Remember that deleting an index is irreversible, destroying physical storage for the fields collection and other constructs. Pause to think about the implications before dropping it.

4. Create a revised index, where the body of the request includes changed or modified field definitions.

5. Load the index with documents from an external source.

When you create the index, physical storage is allocated for each field in the index schema, with an inverted index created for each searchable field. Fields that aren't searchable can be used in filters or expressions, but don't have inverted indexes and aren't full-text or fuzzy searchable. On an index rebuild, these inverted indexes are deleted and recreated based on the index schema you provide.

When you load the index, each field's inverted index is populated with all of the unique, tokenized words from each document, with a map to corresponding document IDs. For example, when indexing a hotels data set, an inverted index created for a City field might contain terms for Seattle, Portland, and so forth. Documents that include Seattle or Portland in the City field would have their document ID listed alongside the term. On any Add, Update or Delete operation, the terms and document ID list are updated accordingly.

Balancing workloads

Indexing doesn't run in the background, but the search service will balance any indexing jobs against ongoing queries. During indexing, you can monitor query requests in the portal to ensure queries are completing in a timely manner.

If indexing workloads introduce unacceptable levels of query latency, conduct performance analysis and review these performance tips for potential mitigation.

Check for updates

You can begin querying an index as soon as the first document is loaded. If you know a document's ID, the Lookup Document REST API returns the specific document. For broader testing, you should wait until the index is fully loaded, and then use queries to verify the context you expect to see.

You can use Search Explorer or a Web testing tool like Postman to check for updated content.

If you added or renamed a field, use $select to return that field:
`search=*&$select=document-id,my-new-field,some-old-field&$count=true`

See also

- Indexer overview
- Index large data sets at scale
- Indexing in the portal
- Azure SQL Database indexer
- Azure Cosmos DB indexer
- Azure Blob Storage indexer
- Azure Table Storage indexer
- Security in Azure Cognitive Search

Additional resources

🕮 Documentation

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Create Index (Azure Cognitive Search REST API)

Define an index schema for fields and other constructs in an Azure Cognitive Search index.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Run or reset indexers - Azure Cognitive Search

Run indexers in full, or reset an indexer, skills, or individual documents to refresh all or part of a search index or knowledge store.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Show 5 more

Create an index alias in Azure Cognitive Search

Article • 04/05/2023 • 3 minutes to read

> **Important**
>
> Index aliases are currently in public preview and available under **supplemental terms of use**.

In Azure Cognitive Search, an index alias is a secondary name that can be used to refer to an index for querying, indexing, and other operations. You can create an alias that maps to a search index and substitute the alias name in places where you would otherwise reference an index name. An alias adds flexibility if you need to change which index your application is pointing to. Instead of updating the references in your application, you can just update the mapping for your alias.

The main goal of index aliases is to make it easier to manage your production indexes. For example, if you need to make a change to your index definition, such as editing a field or adding a new analyzer, you'll have to create a new search index because all search indexes are immutable. This means you either need to drop and rebuild your index or create a new index and then migrate your application over to that index.

Instead of dropping and rebuilding your index, you can use index aliases. A typical workflow would be to:

1. Create your search index
2. Create an alias that maps to your search index
3. Have your application send querying/indexing requests to the alias rather than the index name
4. When you need to make a change to your index that requires a rebuild, create a new search index
5. When your new index is ready to go, update the alias to map to the new index and requests will automatically be routed to the new index

Create an index alias

You can create an alias using the preview REST API, the preview SDKs, or through the Azure portal. An alias consists of the `name` of the alias and the name of the search

index that the alias is mapped to. Only one index name can be specified in the `indexes` array.

REST API

You can use the Create or Update Alias (REST preview) to create an index alias.

HTTP

```
POST /aliases?api-version=2021-04-30-preview
{
    "name": "my-alias",
    "indexes": ["hotel-samples-index"]
}
```

Send requests to an index alias

Once you've created your alias, you're ready to start using it. Aliases can be used for all document operations including querying, indexing, suggestions, and autocomplete.

In the query below, instead of sending the request to `hotel-samples-index`, you can instead send the request to `my-alias` and it will be routed accordingly.

HTTP

```
POST /indexes/my-alias/docs/search?api-version=2021-04-30-preview
{
    "search": "pool spa +airport",
    "searchMode": any,
    "queryType": "simple",
    "select": "HotelId, HotelName, Category, Description",
    "count": true
}
```

If you expect to make updates to a production index, specify an alias rather than the index name in your client-side application. Scenarios that require an index rebuild are outlined in Drop and rebuild an index.

> **Note**
>
> You can only use an alias with **document operations** or to get and update an index definition. Aliases can't be used to delete an index, can't be used with the Analyze Text API, and can't be used as the `targetIndexName` on an indexer.

An update to an alias may take up to 10 seconds to propagate through the system so you should wait at least 10 seconds before performing any operation in the index that has been mapped or recently was mapped to the alias.

Swap indexes

Now, whenever you need to update your application to point to a new index, all you need to do is update the mapping in your alias. PUT is required for updates as described in Create or Update Alias (REST preview).

HTTP

```
PUT /aliases/my-alias?api-version=2021-04-30-preview
{
    "name": "my-alias",
    "indexes": ["hotel-samples-index2"]
}
```

After you make the update to the alias, requests will automatically start to be routed to the new index.

> **Note**
>
> An update to an alias may take up to 10 seconds to propagate through the system so you should wait at least 10 seconds before deleting the index that the alias was previously mapped to.

See also

- Drop and rebuild an index in Azure Cognitive Search

Create an index for multiple languages in Azure Cognitive Search

Article • 01/19/2023 • 5 minutes to read

A multilingual search application is one that provides a search experience in the user's own language. Language support is enabled through a language analyzer assigned to string field. Cognitive Search supports Microsoft and Lucene analyzers. The language analyzer determines the linguistic rules by which content is tokenized. By default, the search engine uses Standard Lucene, which is language agnostic. If testing shows that the default analyzer is insufficient, replace it with a language analyzer.

In Azure Cognitive Search, the two patterns for supporting a multi-lingual audience include:

- Create language-specific indexes where all of the alphanumeric content is in the same language, and all searchable string fields are attributed to use the same language analyzer.
- Create a blended index with language-specific versions of each field (for example, description_en, description_fr, description_ko), and then constrain full text search to just those fields at query time. This approach is useful for scenarios where language variants are only needed on a few fields, like a description.

This article focuses on best practices for defining and querying language specific fields in a blended index. The steps you'll implement include:

- ✓ Define a string field for each language variant.
- ✓ Set a language analyzer on each field.
- ✓ On the query request, set the `searchFields` parameter to specific fields, and then use `select` to return just those fields that have compatible content.

Prerequisites

Language analysis applies to fields of type `Edm.String` that are `searchable`, and that contain localized text. If you also need text translation, review the next section to see if AI enrichment fits your scenario.

Non-string fields and non-searchable string fields don't undergo lexical analysis and aren't tokenized. Instead, they are stored and returned verbatim.

Add text translation

This article assumes you have translated strings in place. If that's not the case, you can attach Cognitive Services to an enrichment pipeline, invoking text translation during data ingestion. Text translation takes a dependency on the indexer feature and Cognitive Services, but all setup is done within Azure Cognitive Search.

To add text translation, follow these steps:

1. Verify your content is in a supported data source.
2. Create a data source that points to your content.
3. Create a skillset that includes the Text Translation skill.

   The Text Translation skill takes a single string as input. If you have multiple fields, can create a skillset that calls Text Translation multiple times, once for each field. Alternatively, you can use the Text Merger skill to consolidate the content of multiple fields into one long string.
4. Create an index that includes fields for translated strings. Most of this article covers index design and field definitions for indexing and querying multi-language content.
5. Attach a multi-region Cognitive Services resource to your skillset.
6. Create and run the indexer, and then apply the guidance in this article to query just the fields of interest.

> **Tip**
>
> Text translation is built into the **Import data wizard**. If you have a **supported data source** with text you'd like to translate, you can step through the wizard to try out the language detection and translation functionality before writing any code.

Define fields for content in different languages

In Azure Cognitive Search, queries target a single index. Developers who want to provide language-specific strings in a single search experience typically define dedicated fields to store the values: one field for English strings, one for French, and so on.

The "analyzer" property on a field definition is used to set the language analyzer. It will be used for both indexing and query execution.

JSON

```
{
  "name": "hotels-sample-index",
  "fields": [
    {
      "name": "Description",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "en.microsoft"
    },
    {
      "name": "Description_fr",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "fr.microsoft"
    },
```

Build and load an index

An intermediate step is building and populating the index before formulating a query. We mention this step here for completeness. One way to determine index availability is by checking the indexes list in the portal.

Constrain the query and trim results

Parameters on the query are used to limit search to specific fields and then trim the results of any fields not helpful to your scenario.

Parameters	Purpose
searchFields	Limits full text search to the list of named fields.
$select	Trims the response to include only the fields you specify. By default, all retrievable fields are returned. The **$select** parameter lets you choose which ones to return.

Given a goal of constraining search to fields containing French strings, you would use **searchFields** to target the query at fields containing strings in that language.

Specifying the analyzer on a query request isn't necessary. A language analyzer on the field definition will always be used during query processing. For queries that specify multiple fields invoking different language analyzers, the terms or phrases will be processed independently by the assigned analyzers for each field.

By default, a search returns all fields that are marked as retrievable. As such, you might want to exclude fields that don't conform to the language-specific search experience you want to provide. Specifically, if you limited search to a field with French strings, you probably want to exclude fields with English strings from your results. Using the **$select** query parameter gives you control over which fields are returned to the calling application.

Example in REST

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "animaux acceptés",
    "searchFields": "Tags, Description_fr",
    "select": "HotelName, Description_fr, Address/City, Address/StateProvince, Tags",
    "count": "true"
}
```

Example in C#

C#

```
private static void RunQueries(SearchClient srchclient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    options = new SearchOptions()
    {
        IncludeTotalCount = true,
        Filter = "",
        OrderBy = { "" }
    };

    options.Select.Add("HotelId");
    options.Select.Add("HotelName");
    options.Select.Add("Description_fr");
    options.SearchFields.Add("Tags");
    options.SearchFields.Add("Description_fr");

    response = srchclient.Search<Hotel>("*", options);
    WriteDocuments(response);
}
```

Boost language-specific fields

Sometimes the language of the agent issuing a query isn't known, in which case the query can be issued against all fields simultaneously. IA preference for results in a certain language can be defined using scoring profiles. In the example below, matches found in the description in French will be scored higher relative to matches in other languages:

JSON

```
  "scoringProfiles": [
    {
      "name": "frenchFirst",
      "text": {
        "weights": { "description_fr": 2 }
      }
    }
  ]
```

You would then include the scoring profile in the search request:

HTTP

```
POST /indexes/hotels/docs/search?api-version=2020-06-30
{
  "search": "pets allowed",
  "searchFields": "Tags, Description_fr",
  "select": "HotelName, Tags, Description_fr",
  "scoringProfile": "frenchFirst",
  "count": "true"
}
```

Next steps

- Add a language analyzer
- How full text search works in Azure Cognitive Search
- Search Documents REST API
- AI enrichment overview
- Skillsets overview

Additional resources

🕮 Documentation

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Add language analyzers to string fields - Azure Cognitive Search

Multi-lingual lexical analysis for non-English queries and indexes in Azure Cognitive Search.

Configure semantic search - Azure Cognitive Search

Set a semantic query type to attach the deep learning models of semantic search.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Cognitive Search FAQ - Azure Cognitive Search

Get answers to common questions about Microsoft Azure Cognitive Search service, a cloud hosted search service on Microsoft Azure.

Semantic search - Azure Cognitive Search

Learn how Cognitive Search is using deep learning semantic search models from Bing to make search results more intuitive.

C# tutorial optimize indexing with the push API - Azure Cognitive Search

Learn how to efficiently index data using Azure Cognitive Search's push API. This tutorial and sample code are in C#.

Show 5 more

Index large data sets in Azure Cognitive Search

Article • 02/17/2023 • 7 minutes to read

If your search solution requirements include indexing big data or complex data, this article describes the strategies for accommodating long running processes on Azure Cognitive Search.

This article assumes familiarity with the two basic approaches for importing data: pushing data into an index, or pulling in data from a supported data source using a search indexer. The strategy you choose will be determined by the indexing approach you're already using. If your scenario involves computationally intensive AI enrichment, then your strategy must include indexers, given the skillset dependency on indexers.

This article complements Tips for better performance, which offers best practices on index and query design. A well-designed index that includes only the fields and attributes you need is an important prerequisite for large-scale indexing.

> ⓘ **Note**
>
> The strategies described in this article assume a single large data source. If your solution requires indexing from multiple data sources, see **Index multiple data sources in Azure Cognitive Search** for a recommended approach.

Index large data using the push APIs

"Push" APIs, such as Add Documents REST API or the IndexDocuments method (Azure SDK for .NET), are the most prevalent form of indexing in Cognitive Search. For solutions that use a push API, the strategy for long-running indexing will have one or both of the following components:

- Batching documents
- Managing threads

Batch multiple documents per request

A simple mechanism for indexing a large quantity of data is to submit multiple documents or records in a single request. As long as the entire payload is under 16 MB, a request can handle up to 1000 documents in a bulk upload operation. These limits

apply whether you're using the Add Documents REST API or the IndexDocuments method in the .NET SDK. For either API, you would package 1000 documents in the body of each request.

Batching documents will significantly shorten the amount of time it takes to work through a large data volume. Determining the optimal batch size for your data is a key component of optimizing indexing speeds. The two primary factors influencing the optimal batch size are:

- The schema of your index
- The size of your data

Because the optimal batch size depends on your index and your data, the best approach is to test different batch sizes to determine which one results in the fastest indexing speeds for your scenario. Tutorial: Optimize indexing with the push API provides sample code for testing batch sizes using the .NET SDK.

Add threads and a retry strategy

Indexers have built-in thread management, but when you're using the push APIs, your application code will have to manage threads. Make sure there are sufficient threads to make full use of the available capacity, especially if you've recently increased partitions or have upgraded to a higher tier search service.

1. Increase the number of concurrent threads in your client code.

2. As you ramp up the requests hitting the search service, you might encounter HTTP status codes indicating the request didn't fully succeed. During indexing, two common HTTP status codes are:

    - **503 Service Unavailable** - This error means that the system is under heavy load and your request can't be processed at this time.

    - **207 Multi-Status** - This error means that some documents succeeded, but at least one failed.

3. To handle failures, requests should be retried using an exponential backoff retry strategy.

The Azure .NET SDK automatically retries 503s and other failed requests, but you'll need to implement your own logic to retry 207s. Open-source tools such as Polly can also be used to implement a retry strategy.

Index with indexers and the "pull" APIs

Indexers have several capabilities that are useful for long-running processes:

- Batching documents
- Parallel indexing over partitioned data
- Scheduling and change detection for indexing just new and change documents over time

Indexer schedules can resume processing at the last known stopping point. If data isn't fully indexed within the processing window, the indexer picks up wherever it left off on the next run, assuming you're using a data source that provides change detection.

Partitioning data into smaller individual data sources enables parallel processing. You can break up source data, such as into multiple containers in Azure Blob Storage, create a data source for each partition, and then run the indexers in parallel, subject to the number of search units of your search service.

Check indexer batch size

As with the push API, indexers allow you to configure the number of items per batch. For indexers based on the Create Indexer REST API, you can set the `batchSize` argument to customize this setting to better match the characteristics of your data.

Default batch sizes are data source specific. Azure SQL Database and Azure Cosmos DB have a default batch size of 1000. In contrast, Azure Blob indexing sets batch size at 10 documents in recognition of the larger average document size.

Schedule indexers for long-running processes

Indexer scheduling is an important mechanism for processing large data sets and for accommodating slow-running processes like image analysis in an enrichment pipeline.

Typically, indexer processing runs within a 2-hour window. If the indexing workload takes days rather than hours to complete, you can put the indexer on a consecutive, recurring schedule that starts every two hours. Assuming the data source has change tracking enabled, the indexer will resume processing where it last left off. At this cadence, an indexer can work its way through a document backlog over a series of days until all unprocessed documents are processed.

```
JSON
```

```
{
    "dataSourceName" : "hotels-ds",
    "targetIndexName" : "hotels-idx",
    "schedule" : { "interval" : "PT2H", "startTime" : "2022-01-01T00:00:00Z"
}
}
```

When there are no longer any new or updated documents in the data source, indexer execution history will report `0/0` documents processed, and no processing occurs.

For more information about setting schedules, see Create Indexer REST API or see How to schedule indexers for Azure Cognitive Search.

> **Note**
>
> Some indexers that run on an older runtime architecture have a 24-hour rather than 2-hour maximum processing window. The 2-hour limit is for newer content processors that run in an **internally managed multi-tenant environment**. Whenever possible, Azure Cognitive Search tries to offload indexer and skillset processing to the multi-tenant environment. If the indexer can't be migrated, it will run in the private environment and it can run for as long as 24 hours. If you're scheduling an indexer that exhibits these characteristics, assume a 24 hour processing window.

Run indexers in parallel

If you partition your data, you can create multiple indexer-data-source combinations that pull from each data source and write to the same search index. Because each indexer is distinct, you can run them at the same time, populating a search index more quickly than if you ran them sequentially.

Make sure you have sufficient capacity. One search unit in your service can run one indexer at any given time. Creating multiple indexers is only useful if they can run in parallel.

The number of indexing jobs that can run simultaneously varies for text-based and skills-based indexing. For more information, see Indexer execution.

If your data source is an Azure Blob Storage container or Azure Data Lake Storage Gen 2, enumerating a large number of blobs can take a long time (even hours) until this operation is completed. This will cause that your indexer's documents succeeded count isn't increased during that time and it may seem it's not making any progress, when it is.

If you would like document processing to go faster for a large number of blobs, consider partitioning your data into multiple containers and create parallel indexers pointing to a single index.

1. Sign in to Azure portal and check the number of search units used by your search service. Select **Settings** > **Scale** to view the number at the top of the page. The number of indexers that will run in parallel is approximately equal to the number of search units.

2. Partition source data among multiple containers or multiple virtual folders inside the same container.

3. Create multiple data sources, one for each partition, paired to its own indexer.

4. Specify the same target search index in each indexer.

5. Schedule the indexers.

6. Review indexer status and execution history for confirmation.

There are some risks associated with parallel indexing. First, recall that indexing doesn't run in the background, increasing the likelihood that queries will be throttled or dropped.

Second, Azure Cognitive Search doesn't lock the index for updates. Concurrent writes are managed, invoking a retry if a particular write doesn't succeed on first attempt, but you might notice an increase in indexing failures.

Although multiple indexer-data-source sets can target the same index, be careful of indexer runs that can overwrite existing values in the index. If a second indexer-data-source targets the same documents and fields, any values from the first run will be overwritten. Field values are replaced in full; an indexer can't merge values from multiple runs into the same field.

Index big data on Spark

If you have a big data architecture and your data is on a Spark cluster, we recommend SynapseML for loading and indexing data. The tutorial includes steps for calling Cognitive Services for AI enrichment, but you can also use the AzureSearchWriter API for text indexing.

See also

- Tips for improving performance
- Performance analysis
- Indexer overview
- Monitor indexer status

Synonyms in Azure Cognitive Search

Article • 09/13/2022 • 6 minutes to read

Within a search service, synonym maps are a global resource that associate equivalent terms, expanding the scope of a query without the user having to actually provide the term. For example, assuming "dog", "canine", and "puppy" are mapped synonyms, a query on "canine" will match on a document containing "dog".

Create synonyms

A synonym map is an asset that can be created once and used by many indexes. The service tier determines how many synonym maps you can create, ranging from three synonym maps for Free and Basic tiers, up to 20 for the Standard tiers.

You might create multiple synonym maps for different languages, such as English and French versions, or lexicons if your content includes technical or obscure terminology. Although you can create multiple synonym maps in your search service, within an index, a field definition can only have one synonym map assignment.

A synonym map consists of name, format, and rules that function as synonym map entries. The only format that is supported is `solr`, and the `solr` format determines rule construction.

HTTP

```
POST /synonymmaps?api-version=2020-06-30
{
    "name": "geo-synonyms",
    "format": "solr",
    "synonyms": "
        USA, United States, United States of America\n
        Washington, Wash., WA => WA\n"
}
```

To create a synonym map, do so programmatically (the portal doesn't support synonym map definitions):

- Create Synonym Map (REST API). This reference is the most descriptive.
- SynonymMap class (.NET) and Add Synonyms using C#
- SynonymMap class (Python)
- SynonymMap interface (JavaScript)
- SynonymMap class (Java)

Define rules

Mapping rules adhere to the open-source synonym filter specification of Apache Solr, described in this document: SynonymFilter. The `solr` format supports two kinds of rules:

- equivalency (where terms are equal substitutes in the query)
- explicit mappings (where terms are mapped to one explicit term prior to querying)

Each rule must be delimited by the new line character (`\n`). You can define up to 5,000 rules per synonym map in a free service and 20,000 rules per map in other tiers. Each rule can have up to 20 expansions (or items in a rule). For more information, see Synonym limits.

Query parsers will lower-case any upper or mixed case terms, but if you want to preserve special characters in the string, such as a comma or dash, add the appropriate escape characters when creating the synonym map.

Equivalency rules

Rules for equivalent terms are comma-delimited within the same rule. In the first example, a query on `USA` will expand to `USA` OR `"United States"` OR `"United States of America"`. Notice that if you want to match on a phrase, the query itself must be a quote-enclosed phrase query.

In the equivalence case, a query for `dog` will expand the query to also include `puppy` and `canine`.

JSON

```
{
"format": "solr",
"synonyms": "
    USA, United States, United States of America\n
    dog, puppy, canine\n
    coffee, latte, cup of joe, java\n"
}
```

Explicit mapping

Rules for an explicit mapping are denoted by an arrow `=>`. When specified, a term sequence of a search query that matches the left-hand side of `=>` will be replaced with

the alternatives on the right-hand side at query time.

In the explicit case, a query for `Washington`, `Wash.` or `WA` will be rewritten as `WA`, and the query engine will only look for matches on the term `WA`. Explicit mapping only applies in the direction specified, and doesn't rewrite the query `WA` to `Washington` in this case.

JSON

```
{
"format": "solr",
"synonyms": "
    Washington, Wash., WA => WA\n
    California, Calif., CA => CA\n"
}
```

Escaping special characters

In full text search, synonyms are analyzed during query processing just like any other query term, which means that rules around reserved and special characters apply to the terms in your synonym map. The list of characters that requires escaping varies between the simple syntax and full syntax:

- simple syntax `+ | " ( ) ' \`
- full syntax `+ - & | ! ( ) { } [ ] ^ " ~ * ? : \ /`

Recall that if you need to preserve characters that would otherwise be discarded by the default analyzer during indexing, you should substitute an analyzer that preserves them. Some choices include Microsoft natural language analyzers, which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see Partial terms, patterns, and special characters.

The following example shows an example of how to escape a character with a backslash:

JSON

```
{
"format": "solr",
"synonyms": "WA\, USA, WA, Washington\n"
}
```

Since the backslash is itself a special character in other languages like JSON and C#, you'll probably need to double-escape it. For example, the JSON sent to the REST API for the above synonym map would look like this:

JSON

```
{
"format":"solr",
"synonyms": "WA\\, USA, WA, Washington"
}
```

Upload and manage synonym maps

As mentioned previously, you can create or update a synonym map without disrupting query and indexing workloads. A synonym map is a standalone object (like indexes or data sources), and as long as no field is using it, updates won't cause indexing or queries to fail. However, once you add a synonym map to a field definition, if you then delete a synonym map, any query that includes the fields in question will fail with a 404 error.

Creating, updating, and deleting a synonym map is always a whole-document operation, meaning that you can't update or delete parts of the synonym map incrementally. Updating even a single rule requires a reload.

Assign synonyms to fields

After uploading a synonym map, you can enable the synonyms on fields of the type `Edm.String` or `Collection(Edm.String)`, on fields having `"searchable":true`. As noted, a field definition can use only one synonym map.

HTTP

```
POST /indexes?api-version=2020-06-30
{
    "name":"hotels-sample-index",
    "fields":[
        {
            "name":"description",
            "type":"Edm.String",
            "searchable":true,
            "synonymMaps":[
            "en-synonyms"
            ]
        },
        {
            "name":"description_fr",
            "type":"Edm.String",
            "searchable":true,
            "analyzer":"fr.microsoft",
            "synonymMaps":[
            "fr-synonyms"
```

```
            ]
        }
    ]
}
```

Query on equivalent or mapped fields

Adding synonyms doesn't impose new requirements on query construction. You can issue term and phrase queries just as you did before the addition of synonyms. The only difference is that if a query term exists in the synonym map, the query engine will either expand or rewrite the term or phrase, depending on the rule.

How synonyms are used during query execution

Synonyms are a query expansion technique that supplements the contents of an index with equivalent terms, but only for fields that have a synonym assignment. If a field-scoped query *excludes* a synonym-enabled field, you won't see matches from the synonym map.

For synonym-enabled fields, synonyms are subject to the same text analysis as the associated field. For example, if a field is analyzed using the standard Lucene analyzer, synonym terms will also be subject to the standard Lucene analyzer at query time. If you want to preserve punctuation, such as periods or dashes, in the synonym term, apply a content-preserving analyzer on the field.

Internally, the synonyms feature rewrites the original query with synonyms with the OR operator. For this reason, hit highlighting and scoring profiles treat the original term and synonyms as equivalent.

Synonyms apply to free-form text queries only and aren't supported for filters, facets, autocomplete, or suggestions. Autocomplete and suggestions are based only on the original term; synonym matches don't appear in the response.

Synonym expansions don't apply to wildcard search terms; prefix, fuzzy, and regex terms aren't expanded.

If you need to do a single query that applies synonym expansion and wildcard, regex, or fuzzy searches, you can combine the queries using the OR syntax. For example, to combine synonyms with wildcards for simple query syntax, the term would be `<query> | <query>*`.

If you have an existing index in a development (non-production) environment, experiment with a small dictionary to see how the addition of synonyms changes the search experience, including impact on scoring profiles, hit highlighting, and suggestions.

Next steps

Create a synonym map (REST API)

Example: Add synonyms for Azure Cognitive Search in C#

Article • 06/16/2022 • 4 minutes to read

Synonyms expand a query by matching on terms considered semantically equivalent to the input term. For example, you might want "car" to match documents containing the terms "automobile" or "vehicle".

In Azure Cognitive Search, synonyms are defined in a *synonym map*, through *mapping rules* that associate equivalent terms. This example covers essential steps for adding and using synonyms with an existing index.

In this example, you will learn how to:

- ✓ Create a synonym map using the SynonymMap class.
- ✓ Set the SynonymMapsName property on fields that should support query expansion via synonyms.

You can query a synonym-enabled field as you would normally. There is no additional query syntax required to access synonyms.

You can create multiple synonym maps, post them as a service-wide resource available to any index, and then reference which one to use at the field level. At query time, in addition to searching an index, Azure Cognitive Search does a lookup in a synonym map, if one is specified on fields used in the query.

> **Note**
>
> Synonyms can be created programmatically, but not in the portal.

Prerequisites

Tutorial requirements include the following:

- Visual Studio
- Azure Cognitive Search service
- Azure.Search.Documents package

If you are unfamiliar with the .NET client library, see How to use Azure Cognitive Search in .NET.

Sample code

You can find the full source code of the sample application used in this example on GitHub.

Overview

Before-and-after queries are used to demonstrate the value of synonyms. In this example, a sample application executes queries and returns results on a sample "hotels" index populated with two documents. First, the application executes search queries using terms and phrases that do not appear in the index. Second, the code enables the synonyms feature, then re-issues the same queries, this time returning results based on matches in the synonym map.

The code below demonstrates the overall flow.

C#

```
static void Main(string[] args)
{
   SearchIndexClient indexClient = CreateSearchIndexClient();

   Console.WriteLine("Cleaning up resources...\n");
   CleanupResources(indexClient);

   Console.WriteLine("Creating index...\n");
   CreateHotelsIndex(indexClient);

   SearchClient searchClient = indexClient.GetSearchClient("hotels");

   Console.WriteLine("Uploading documents...\n");
   UploadDocuments(searchClient);

   SearchClient searchClientForQueries = CreateSearchClientForQueries();

   RunQueriesWithNonExistentTermsInIndex(searchClientForQueries);

   Console.WriteLine("Adding synonyms...\n");
   UploadSynonyms(indexClient);

   Console.WriteLine("Enabling synonyms in the test index...\n");
   EnableSynonymsInHotelsIndexSafely(indexClient);
   Thread.Sleep(10000); // Wait for the changes to propagate

   RunQueriesWithNonExistentTermsInIndex(searchClientForQueries);

   Console.WriteLine("Complete.  Press any key to end application...\n");
```

```
    Console.ReadKey();
}
```

"Before" queries

In `RunQueriesWithNonExistentTermsInIndex`, issue search queries with "five star", "internet", and "economy AND hotel".

Phrase queries, such as "five star", must be enclosed in quotation marks, and might also need escape characters depending on your client.

Bash

```
Console.WriteLine("Search the entire index for the phrase \"five
star\":\n");
results = searchClient.Search<Hotel>("\"five star\"", searchOptions);
WriteDocuments(results);

Console.WriteLine("Search the entire index for the term 'internet':\n");
results = searchClient.Search<Hotel>("internet", searchOptions);
WriteDocuments(results);

Console.WriteLine("Search the entire index for the terms 'economy' AND
'hotel':\n");
results = searchClient.Search<Hotel>("economy AND hotel", searchOptions);
WriteDocuments(results);
```

Neither of the two indexed documents contain the terms, so we get the following output from the first `RunQueriesWithNonExistentTermsInIndex`: **no document matched**.

Enable synonyms

After the "before" queries are run, the sample code enables synonyms. Enabling synonyms is a two-step process. First, define and upload synonym rules. Second, configure fields to use them. The process is outlined in `UploadSynonyms` and `EnableSynonymsInHotelsIndex`.

1. Add a synonym map to your search service. In `UploadSynonyms`, we define four rules in our synonym map 'desc-synonymmap' and upload to the service.

   C#

   ```
   private static void UploadSynonyms(SearchIndexClient indexClient)
   {
      var synonymMap = new SynonymMap("desc-synonymmap", "hotel,
   ```

```
motel\ninternet,wifi\nfive star=>luxury\neconomy,inexpensive=>budget");

    indexClient.CreateOrUpdateSynonymMap(synonymMap);
}
```

2. Configure searchable fields to use the synonym map in the index definition. In `AddSynonymMapsToFields`, we enable synonyms on two fields `category` and `tags` by setting the `SynonymMapNames` property to the name of the newly uploaded synonym map.

```C#
private static SearchIndex AddSynonymMapsToFields(SearchIndex index)
{
    index.Fields.First(f => f.Name ==
"category").SynonymMapNames.Add("desc-synonymmap");
    index.Fields.First(f => f.Name == "tags").SynonymMapNames.Add("desc-
synonymmap");
    return index;
}
```

When you add a synonym map, index rebuilds are not required. You can add a synonym map to your service, and then amend existing field definitions in any index to use the new synonym map. The addition of new attributes has no impact on index availability. The same applies in disabling synonyms for a field. You can simply set the `SynonymMapNames` property to an empty list.

```C#
index.Fields.First(f => f.Name ==
"category").SynonymMapNames.Add("desc-synonymmap");
```

"After" queries

After the synonym map is uploaded and the index is updated to use the synonym map, the second `RunQueriesWithNonExistentTermsInIndex` call outputs the following:

```Bash
Search the entire index for the phrase "five star":

Name: Fancy Stay        Category: Luxury        Tags: [pool, view, wifi,
concierge]

Search the entire index for the term 'internet':
```

```
Name: Fancy Stay        Category: Luxury        Tags: [pool, view, wifi,
concierge]

Search the entire index for the terms 'economy' AND 'hotel':

Name: Roach Motel       Category: Budget        Tags: [motel, budget]
```

The first query finds the document from the rule `five star=>luxury`. The second query expands the search using `internet,wifi` and the third using both `hotel, motel` and `economy,inexpensive=>budget` in finding the documents they matched.

Adding synonyms completely changes the search experience. In this example, the original queries failed to return meaningful results even though the documents in our index were relevant. By enabling synonyms, we can expand an index to include terms in common use, with no changes to underlying data in the index.

Clean up resources

The fastest way to clean up after an example is by deleting the resource group containing the Azure Cognitive Search service. You can delete the resource group now to permanently delete everything in it. In the portal, the resource group name is on the Overview page of Azure Cognitive Search service.

Next steps

This example demonstrated the synonyms feature in C# code to create and post mapping rules and then call the synonym map on a query. Additional information can be found in the .NET SDK and REST API reference documentation.

How to use synonyms in Azure Cognitive Search

Model complex data types in Azure Cognitive Search

Article • 01/31/2023 • 10 minutes to read

External datasets used to populate an Azure Cognitive Search index can come in many shapes. Sometimes they include hierarchical or nested substructures. Examples might include multiple addresses for a single customer, multiple colors and sizes for a single SKU, multiple authors of a single book, and so on. In modeling terms, you might see these structures referred to as *complex*, *compound*, *composite*, or *aggregate* data types. The term Azure Cognitive Search uses for this concept is **complex type**. In Azure Cognitive Search, complex types are modeled using **complex fields**. A complex field is a field that contains children (sub-fields) which can be of any data type, including other complex types. This works in a similar way as structured data types in a programming language.

Complex fields represent either a single object in the document, or an array of objects, depending on the data type. Fields of type `Edm.ComplexType` represent single objects, while fields of type `Collection(Edm.ComplexType)` represent arrays of objects.

Azure Cognitive Search natively supports complex types and collections. These types allow you to model almost any JSON structure in an Azure Cognitive Search index. In previous versions of Azure Cognitive Search APIs, only flattened row sets could be imported. In the newest version, your index can now more closely correspond to source data. In other words, if your source data has complex types, your index can have complex types also.

To get started, we recommend the Hotels data set, which you can load in the **Import data** wizard in the Azure portal. The wizard detects complex types in the source and suggests an index schema based on the detected structures.

> **Note**
>
> Support for complex types became generally available starting in `api-version=2019-05-06`.
>
> If your search solution is built on earlier workarounds of flattened datasets in a collection, you should change your index to include complex types as supported in the newest API version. For more information about upgrading API versions, see **Upgrade to the newest REST API version** or **Upgrade to the newest .NET SDK version**.

Example of a complex structure

The following JSON document is composed of simple fields and complex fields. Complex fields, such as `Address` and `Rooms`, have sub-fields. `Address` has a single set of values for those sub-fields, since it's a single object in the document. In contrast, `Rooms` has multiple sets of values for its sub-fields, one for each object in the collection.

JSON

```
{
  "HotelId": "1",
  "HotelName": "Secret Point Motel",
  "Description": "Ideally located on the main commercial artery of the city
in the heart of New York.",
  "Tags": ["Free wifi", "on-site parking", "indoor pool", "continental
breakfast"],
  "Address": {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY"
  },
  "Rooms": [
    {
      "Description": "Budget Room, 1 Queen Bed (Cityside)",
      "RoomNumber": 1105,
      "BaseRate": 96.99,
    },
    {
      "Description": "Deluxe Room, 2 Double Beds (City View)",
      "Type": "Deluxe Room",
      "BaseRate": 150.99,
    }
    . . .
  ]
}
```

Indexing complex types

During indexing, you can have a maximum of 3000 elements across all complex collections within a single document. An element of a complex collection is a member of that collection, so in the case of Rooms (the only complex collection in the Hotel example), each room is an element. In the example above, if the "Secret Point Motel" had 500 rooms, the hotel document would have 500 room elements. For nested complex collections, each nested element is also counted, in addition to the outer (parent) element.

This limit applies only to complex collections, and not complex types (like Address) or string collections (like Tags).

Create complex fields

As with any index definition, you can use the portal, REST API, or .NET SDK to create a schema that includes complex types.

Other Azure SDKs provide samples in Python, Java, and JavaScript.

Azure portal

1. Sign in to Azure portal.
2. On the search service **Overview** page, select the **Indexes** tab.
3. Open an existing index or create a new index.
4. Select the **Fields** tab, and then select **Add field**. An empty field is added. If you're working with an existing fields collection, scroll down to set up the field.
5. Give the field a name and set the type to either `Edm.ComplexType` or `Collection(Edm.ComplexType)`.
6. Select the ellipses on the far right, and then select either **Add field** or **Add subfield**, and then assign attributes.

Update complex fields

All of the reindexing rules that apply to fields in general still apply to complex fields. Restating a few of the main rules here, adding a field to a complex type doesn't require an index rebuild, but most modifications do.

Structural updates to the definition

You can add new sub-fields to a complex field at any time without the need for an index rebuild. For example, adding "ZipCode" to `Address` or "Amenities" to `Rooms` is allowed, just like adding a top-level field to an index. Existing documents have a null value for new fields until you explicitly populate those fields by updating your data.

Notice that within a complex type, each sub-field has a type and may have attributes, just as top-level fields do

Data updates

Updating existing documents in an index with the `upload` action works the same way for complex and simple fields -- all fields are replaced. However, `merge` (or `mergeOrUpload` when applied to an existing document) doesn't work the same across all fields. Specifically, `merge` doesn't support merging elements within a collection. This limitation exists for collections of primitive types and complex collections. To update a collection, you'll need to retrieve the full collection value, make changes, and then include the new collection in the Index API request.

Search complex fields

Free-form search expressions work as expected with complex types. If any searchable field or sub-field anywhere in a document matches, then the document itself is a match.

Queries get more nuanced when you have multiple terms and operators, and some terms have field names specified, as is possible with the Lucene syntax. For example, this query attempts to match two terms, "Portland" and "OR", against two sub-fields of the Address field:

```
search=Address/City:Portland AND Address/State:OR
```

Queries like this are *uncorrelated* for full-text search, unlike filters. In filters, queries over sub-fields of a complex collection are correlated using range variables in any or all. The Lucene query above returns documents containing both "Portland, Maine" and "Portland, Oregon", along with other cities in Oregon. This happens because each clause applies to all values of its field in the entire document, so there's no concept of a "current sub-document". For more information on this, see Understanding OData collection filters in Azure Cognitive Search.

Select complex fields

The `$select` parameter is used to choose which fields are returned in search results. To use this parameter to select specific sub-fields of a complex field, include the parent field and sub-field separated by a slash (`/`).

```
$select=HotelName, Address/City, Rooms/BaseRate
```

Fields must be marked as Retrievable in the index if you want them in search results. Only fields marked as Retrievable can be used in a `$select` statement.

Filter, facet, and sort complex fields

The same OData path syntax used for filtering and fielded searches can also be used for faceting, sorting, and selecting fields in a search request. For complex types, rules apply that govern which sub-fields can be marked as sortable or facetable. For more information on these rules, see the Create Index API reference.

Faceting sub-fields

Any sub-field can be marked as facetable unless it is of type `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)`.

The document counts returned in the facet results are calculated for the parent document (a hotel), not the sub-documents in a complex collection (rooms). For example, suppose a hotel has 20 rooms of type "suite". Given this facet parameter `facet=Rooms/Type`, the facet count will be one for the hotel, not 20 for the rooms.

Sorting complex fields

Sort operations apply to documents (Hotels) and not sub-documents (Rooms). When you have a complex type collection, such as Rooms, it's important to realize that you can't sort on Rooms at all. In fact, you can't sort on any collection.

Sort operations work when fields have a single value per document, whether the field is a simple field, or a sub-field in a complex type. For example, `Address/City` is allowed to be sortable because there's only one address per hotel, so `$orderby=Address/City` will sort hotels by city.

Filtering on complex fields

You can refer to sub-fields of a complex field in a filter expression. Just use the same OData path syntax that's used for faceting, sorting, and selecting fields. For example, the following filter will return all hotels in Canada:

> `$filter=Address/Country eq 'Canada'`

To filter on a complex collection field, you can use a **lambda expression** with the any and all operators. In that case, the **range variable** of the lambda expression is an object with sub-fields. You can refer to those sub-fields with the standard OData path syntax. For example, the following filter will return all hotels with at least one deluxe room and all non-smoking rooms:

```
$filter=Rooms/any(room: room/Type eq 'Deluxe Room') and Rooms/all(room: not room/SmokingAllowed)
```

As with top-level simple fields, simple sub-fields of complex fields can only be included in filters if they have the **filterable** attribute set to `true` in the index definition. For more information, see the Create Index API reference.

Next steps

Try the Hotels data set in the **Import data** wizard. You'll need the Azure Cosmos DB connection information provided in the readme to access the data.

With that information in hand, your first step in the wizard is to create a new Azure Cosmos DB data source. Further on in the wizard, when you get to the target index page, you'll see an index with complex types. Create and load this index, and then execute queries to understand the new structure.

Quickstart: portal wizard for import, indexing, and queries

How to model relational SQL data for import and indexing in Azure Cognitive Search

Article • 02/23/2023 • 6 minutes to read

Azure Cognitive Search accepts a flat rowset as input to the indexing pipeline. If your source data originates from joined tables in a SQL Server relational database, this article explains how to construct the result set, and how to model a parent-child relationship in an Azure Cognitive Search index.

As an illustration, we refer to a hypothetical hotels database, based on demo data. Assume the database consists of a Hotels$ table with 50 hotels, and a Rooms$ table with rooms of varying types, rates, and amenities, for a total of 750 rooms. There's a one-to-many relationship between the tables. In our approach, a view provides the query that returns 50 rows, one row per hotel, with associated room detail embedded into each row.

The problem of denormalized data

One of the challenges in working with one-to-many relationships is that standard queries built on joined tables return denormalized data, which doesn't work well in an Azure Cognitive Search scenario. Consider the following example that joins hotels and rooms.

```sql
SELECT * FROM Hotels$
INNER JOIN Rooms$
ON Rooms$.HotelID = Hotels$.HotelID
```

Results from this query return all of the Hotel fields, followed by all Room fields, with preliminary hotel information repeating for each room value.

Results | Messages

	Fields from Hotels$				Fields from Rooms$				
	HotelID	HotelName	Description	State	HotelID	Description	Type	BaseRate	BedOptions
181	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (City View)	Standard Room	121.99	1 Queen Bed
182	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 1 King Bed (Waterfront View)	Budget Room	88.99	1 King Bed
183	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 2 Double Beds (Cityside)	Standard Room	127.99	2 Double Beds
184	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 2 Double Beds (Cityside)	Budget Room	96.99	2 Double Beds
185	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Budget Room, 1 Queen Bed (Mountain View)	Budget Room	63.99	1 Queen Bed
186	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (Mountain View)	Standard Room	124.99	1 Queen Bed
187	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	2	Standard Room, 1 Queen Bed (Cityside)	Standard Room	117.99	1 Queen Bed

While this query succeeds on the surface (providing all of the data in a flat row set), it fails in delivering the right document structure for the expected search experience. During indexing, Azure Cognitive Search creates one search document for each row ingested. If your search documents looked like the above results, you would have perceived duplicates - seven separate documents for the Twin Dome hotel alone. A query on "hotels in Florida" would return seven results for just the Twin Dome hotel, pushing other relevant hotels deep into the search results.

To get the expected experience of one document per hotel, you should provide a rowset at the right granularity, but with complete information. This article explains how.

Define a query that returns embedded JSON

To deliver the expected search experience, your data set should consist of one row for each search document in Azure Cognitive Search. In our example, we want one row for each hotel, but we also want our users to be able to search on other room-related fields they care about, such as the nightly rate, size and number of beds, or a view of the beach, all of which are part of a room detail.

The solution is to capture the room detail as nested JSON, and then insert the JSON structure into a field in a view, as shown in the second step.

1. Assume you've two joined tables, Hotels$ and Rooms$, that contain details for 50 hotels and 750 rooms and are joined on the HotelID field. Individually, these tables contain 50 hotels and 750 related rooms.

   ```sql
   SQL
   ```

```sql
CREATE TABLE [dbo].[Hotels$](
  [HotelID] [nchar](10) NOT NULL,
  [HotelName] [nvarchar](255) NULL,
  [Description] [nvarchar](max) NULL,
  [Description_fr] [nvarchar](max) NULL,
  [Category] [nvarchar](255) NULL,
  [Tags] [nvarchar](255) NULL,
  [ParkingIncluded] [float] NULL,
  [SmokingAllowed] [float] NULL,
  [LastRenovationDate] [smalldatetime] NULL,
  [Rating] [float] NULL,
  [StreetAddress] [nvarchar](255) NULL,
  [City] [nvarchar](255) NULL,
  [State] [nvarchar](255) NULL,
  [ZipCode] [nvarchar](255) NULL,
  [GeoCoordinates] [nvarchar](255) NULL
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO

CREATE TABLE [dbo].[Rooms$](
  [HotelID] [nchar](10) NULL,
  [Description] [nvarchar](255) NULL,
  [Description_fr] [nvarchar](255) NULL,
  [Type] [nvarchar](255) NULL,
  [BaseRate] [float] NULL,
  [BedOptions] [nvarchar](255) NULL,
  [SleepsCount] [float] NULL,
  [SmokingAllowed] [float] NULL,
  [Tags] [nvarchar](255) NULL
) ON [PRIMARY]
GO
```

2. Create a view composed of all fields in the parent table (`SELECT * from dbo.Hotels$`), with the addition of a new *Rooms* field that contains the output of a nested query. A **FOR JSON AUTO** clause on `SELECT * from dbo.Rooms$` structures the output as JSON.

SQL

```sql
CREATE VIEW [dbo].[HotelRooms]
AS
SELECT *, (SELECT *
         FROM dbo.Rooms$
         WHERE dbo.Rooms$.HotelID = dbo.Hotels$.HotelID FOR JSON AUTO)
AS Rooms
FROM dbo.Hotels$
GO
```

The following screenshot shows the resulting view, with the *Rooms* nvarchar field at the bottom. The *Rooms* field exists only in the HotelRooms view.

3. Run `SELECT * FROM dbo.HotelRooms` to retrieve the row set. This query returns 50 rows, one per hotel, with associated room information as a JSON collection.

This rowset is now ready for import into Azure Cognitive Search.

> **Note**
>
> This approach assumes that embedded JSON is under the **maximum column size limits of SQL Server**.

Use a complex collection for the "many" side of a one-to-many relationship

On the Azure Cognitive Search side, create an index schema that models the one-to-many relationship using nested JSON. The result set you created in the previous section

generally corresponds to the index schema provided below (we cut some fields for brevity).

The following example is similar to the example in How to model complex data types. The *Rooms* structure, which has been the focus of this article, is in the fields collection of an index named *hotels*. This example also shows a complex type for *Address*, which differs from *Rooms* in that it's composed of a fixed set of items, as opposed to the multiple, arbitrary number of items allowed in a collection.

JSON

```
{
  "name": "hotels",
  "fields": [
    { "name": "HotelId", "type": "Edm.String", "key": true, "filterable":
true },
    { "name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false },
    { "name": "Description", "type": "Edm.String", "searchable": true,
"analyzer": "en.lucene" },
    { "name": "Description_fr", "type": "Edm.String", "searchable": true,
"analyzer": "fr.lucene" },
    { "name": "Category", "type": "Edm.String", "searchable": true,
"filterable": true, "facetable": true },
    { "name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true,
"facetable": true },
    { "name": "Tags", "type": "Collection(Edm.String)", "searchable": true,
"filterable": true, "facetable": true },
    { "name": "Address", "type": "Edm.ComplexType",
      "fields": [
        { "name": "StreetAddress", "type": "Edm.String", "filterable":
false, "sortable": false, "facetable": false, "searchable": true },
        { "name": "City", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true },
        { "name": "StateProvince", "type": "Edm.String", "searchable": true,
"filterable": true, "sortable": true, "facetable": true }
      ]
    },
    { "name": "Rooms", "type": "Collection(Edm.ComplexType)",
      "fields": [
        { "name": "Description", "type": "Edm.String", "searchable": true,
"analyzer": "en.lucene" },
        { "name": "Description_fr", "type": "Edm.String", "searchable":
true, "analyzer": "fr.lucene" },
        { "name": "Type", "type": "Edm.String", "searchable": true },
        { "name": "BaseRate", "type": "Edm.Double", "filterable": true,
"facetable": true },
        { "name": "BedOptions", "type": "Edm.String", "searchable": true,
"filterable": true, "facetable": false },
        { "name": "SleepsCount", "type": "Edm.Int32", "filterable": true,
"facetable": true },
        { "name": "SmokingAllowed", "type": "Edm.Boolean", "filterable":
```

```
true, "facetable": false},
        { "name": "Tags", "type": "Edm.Collection", "searchable": true }
      ]
    }
  ]
}
```

Given the previous result set and the above index schema, you've all the required components for a successful indexing operation. The flattened data set meets indexing requirements yet preserves detail information. In the Azure Cognitive Search index, search results will fall easily into hotel-based entities, while preserving the context of individual rooms and their attributes.

Facet behavior on complex type subfields

Fields that have a parent, such as the fields under Address and Rooms, are called *subfields*. Although you can assign a "facetable" attribute to a subfield, the count of the facet will always be for the main document.

For complex types like Address, where there's just one "Address/City" or "Address/stateProvince" in the document, the facet behavior works as expected. However, in the case of Rooms, where there are multiple subdocuments for each main document, the facet counts can be misleading.

As noted in Model complex types: "the document counts returned in the facet results are calculated for the parent document (a hotel), not the subdocuments in a complex collection (rooms). For example, suppose a hotel has 20 rooms of type "suite". Given this facet parameter facet=Rooms/Type, the facet count is one for the hotel, not 20 for the rooms."

Next steps

Using your own data set, you can use the Import data wizard to create and load the index. The wizard detects the embedded JSON collection, such as the one contained in *Rooms*, and infers an index schema that includes a complex type collection.

Try the following quickstart to learn the basic steps of the Import data wizard.

Quickstart: Create a search index using Azure portal

Additional resources

Documentation

Create an index alias - Azure Cognitive Search

Create an alias to define a secondary name that can be used to refer to an index for querying, indexing, and other operations.

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Model complex data types - Azure Cognitive Search

Nested or hierarchical data structures can be modeled in an Azure Cognitive Search index using ComplexType and Collections data types.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure

Cognitive Search index.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Show 5 more

Add language analyzers to string fields in an Azure Cognitive Search index

Article • 06/24/2022 • 4 minutes to read

A *language analyzer* is a specific type of text analyzer that performs lexical analysis using the linguistic rules of the target language. Every searchable string field has an **analyzer** property. If your content consists of translated strings, such as separate fields for English and Chinese text, you could specify language analyzers on each field to access the rich linguistic capabilities of those analyzers.

When to use a language analyzer

You should consider a language analyzer when awareness of word or sentence structure adds value to text parsing. A common example is the association of irregular verb forms ("bring" and "brought) or plural nouns ("mice" and "mouse"). Without linguistic awareness, these strings are parsed on physical characteristics alone, which fails to catch the connection. Since large chunks of text are more likely to have this content, fields consisting of descriptions, reviews, or summaries are good candidates for a language analyzer.

You should also consider language analyzers when content consists of non-Western language strings. While the default analyzer (Standard Lucene) is language-agnostic, the concept of using spaces and special characters (hyphens and slashes) to separate strings is more applicable to Western languages than non-Western ones.

For example, in Chinese, Japanese, Korean (CJK), and other Asian languages, a space isn't necessarily a word delimiter. Consider the following Japanese string. Because it has no spaces, a language-agnostic analyzer would likely analyze the entire string as one token, when in fact the string is actually a phrase.

```
これは私たちの銀河系の中ではもっとも重く明るいクラスの球状星団です。
(This is the heaviest and brightest group of spherical stars in our galaxy.)
```

For the example above, a successful query would have to include the full token, or a partial token using a suffix wildcard, resulting in an unnatural and limiting search experience.

A better experience is to search for individual words: 明るい (Bright), 私たちの (Our), 銀河系 (Galaxy). Using one of the Japanese analyzers available in Cognitive Search is more likely to unlock this behavior because those analyzers are better equipped at splitting the chunk of text into meaningful words in the target language.

Comparing Lucene and Microsoft Analyzers

Azure Cognitive Search supports 35 language analyzers backed by Lucene, and 50 language analyzers backed by proprietary Microsoft natural language processing technology used in Office and Bing.

Some developers might prefer the more familiar, simple, open-source solution of Lucene. Lucene language analyzers are faster, but the Microsoft analyzers have advanced capabilities, such as lemmatization, word decompounding (in languages like German, Danish, Dutch, Swedish, Norwegian, Estonian, Finish, Hungarian, Slovak) and entity recognition (URLs, emails, dates, numbers). If possible, you should run comparisons of both the Microsoft and Lucene analyzers to decide which one is a better fit. You can use Analyze API to see the tokens generated from a given text using a specific analyzer.

Indexing with Microsoft analyzers is on average two to three times slower than their Lucene equivalents, depending on the language. Search performance shouldn't be significantly affected for average size queries.

English analyzers

The default analyzer is Standard Lucene, which works well for English, but perhaps not as well as Lucene's English analyzer or Microsoft's English analyzer.

- Lucene's English analyzer extends the Standard analyzer. It removes possessives (trailing 's) from words, applies stemming as per Porter Stemming algorithm, and removes English stop words.
- Microsoft's English analyzer performs lemmatization instead of stemming. This means it can handle inflected and irregular word forms much better which results in more relevant search results.

How to specify a language analyzer

Set the analyzer during index creation before it's loaded with data.

1. In the field definition, make sure the field is attributed as "searchable" and is of type Edm.String.

2. Set the "analyzer" property to one of the language analyzers from the supported analyzers list.

   The "analyzer" property is the only property that will accept a language analyzer, and it's used for both indexing and queries. Other analyzer-related properties ("searchAnalyzer" and "indexAnalyzer") won't accept a language analyzer.

Language analyzers can't be customized. If an analyzer isn't meeting your requirements, create a custom analyzer with the microsoft_language_tokenizer or microsoft_language_stemming_tokenizer, and then add filters for pre- and post-tokenization processing.

The following example illustrates a language analyzer specification in an index:

JSON

```
{
  "name": "hotels-sample-index",
  "fields": [
    {
      "name": "Description",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "en.microsoft",
      "indexAnalyzer": null,
      "searchAnalyzer": null
    },
    {
      "name": "Description_fr",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "analyzer": "fr.microsoft",
      "indexAnalyzer": null,
      "searchAnalyzer": null
    },
```

For more information about creating an index and setting field properties, see Create Index (REST). For more information about text analysis, see Analyzers in Azure Cognitive Search.

Supported language analyzers

Below is the list of supported languages, with Lucene and Microsoft analyzer names.

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Arabic	ar.microsoft	ar.lucene
Armenian		hy.lucene
Bangla	bn.microsoft	
Basque		eu.lucene
Bulgarian	bg.microsoft	bg.lucene
Catalan	ca.microsoft	ca.lucene
Chinese Simplified	zh-Hans.microsoft	zh-Hans.lucene
Chinese Traditional	zh-Hant.microsoft	zh-Hant.lucene
Croatian	hr.microsoft	
Czech	cs.microsoft	cs.lucene
Danish	da.microsoft	da.lucene
Dutch	nl.microsoft	nl.lucene
English	en.microsoft	en.lucene
Estonian	et.microsoft	
Finnish	fi.microsoft	fi.lucene
French	fr.microsoft	fr.lucene
Galician		gl.lucene
German	de.microsoft	de.lucene
Greek	el.microsoft	el.lucene
Gujarati	gu.microsoft	
Hebrew	he.microsoft	
Hindi	hi.microsoft	hi.lucene
Hungarian	hu.microsoft	hu.lucene
Icelandic	is.microsoft	
Indonesian (Bahasa)	id.microsoft	id.lucene

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Irish		ga.lucene
Italian	it.microsoft	it.lucene
Japanese	ja.microsoft	ja.lucene
Kannada	kn.microsoft	
Korean	ko.microsoft	ko.lucene
Latvian	lv.microsoft	lv.lucene
Lithuanian	lt.microsoft	
Malayalam	ml.microsoft	
Malay (Latin)	ms.microsoft	
Marathi	mr.microsoft	
Norwegian	nb.microsoft	no.lucene
Persian		fa.lucene
Polish	pl.microsoft	pl.lucene
Portuguese (Brazil)	pt-Br.microsoft	pt-Br.lucene
Portuguese (Portugal)	pt-Pt.microsoft	pt-Pt.lucene
Punjabi	pa.microsoft	
Romanian	ro.microsoft	ro.lucene
Russian	ru.microsoft	ru.lucene
Serbian (Cyrillic)	sr-cyrillic.microsoft	
Serbian (Latin)	sr-latin.microsoft	
Slovak	sk.microsoft	
Slovenian	sl.microsoft	
Spanish	es.microsoft	es.lucene
Swedish	sv.microsoft	sv.lucene
Tamil	ta.microsoft	
Telugu	te.microsoft	

Language	Microsoft Analyzer Name	Lucene Analyzer Name
Thai	th.microsoft	th.lucene
Turkish	tr.microsoft	tr.lucene
Ukrainian	uk.microsoft	
Urdu	ur.microsoft	
Vietnamese	vi.microsoft	

All analyzers with names annotated with **Lucene** are powered by Apache Lucene's language analyzers.

See also

- Create an index
- Create a multi-language index
- Create Index (REST API)
- LexicalAnalyzerName Class (Azure SDK for .NET)

Add custom analyzers to string fields in an Azure Cognitive Search index

Article • 06/24/2022 • 22 minutes to read

A *custom analyzer* is a user-defined combination of tokenizer, one or more token filters, and one or more character filters. A custom analyzer is specified within a search index, and then referenced on field definitions that require custom analysis. A custom analyzer is invoked on a per-field basis. Attributes on the field will determine whether it's used for indexing, queries, or both.

In a custom analyzer, character filters prepare the input text before it's processed by the tokenizer (for example, removing markup). Next, the tokenizer breaks text into tokens. Finally, token filters modify the tokens emitted by the tokenizer. For concepts and examples, see Analyzers in Azure Cognitive Search.

Why use a custom analyzer?

A custom analyzer gives you control over the process of converting text into indexable and searchable tokens by allowing you to choose which types of analysis or filtering to invoke, and the order in which they occur.

Create and assign a custom analyzer if none of default (Standard Lucence), built-in, or language analyzers are sufficient for your needs. You might also create a custom analyzer if you want to use a built-in analyzer with custom options. For example, if you wanted to change the maxTokenLength on Standard, you would create a custom analyzer, with a user-defined name, to set that option.

Scenarios where custom analyzers can be helpful include:

- Using character filters to remove HTML markup before text inputs are tokenized, or replace certain characters or symbols.
- Phonetic search. Add a phonetic filter to enable searching based on how a word sounds, not how it’s spelled.
- Disable lexical analysis. Use the Keyword analyzer to create searchable fields that aren't analyzed.
- Fast prefix/suffix search. Add the Edge N-gram token filter to index prefixes of words to enable fast prefix matching. Combine it with the Reverse token filter to do suffix matching.

- Custom tokenization. For example, use the Whitespace tokenizer to break sentences into tokens using whitespace as a delimiter
- ASCII folding. Add the Standard ASCII folding filter to normalize diacritics like ö or ê in search terms.

> **Note**
>
> Custom analyzers aren't exposed in the Azure portal. The only way to add a custom analyzer is through code that defines an index.

Create a custom analyzer

To create a custom analyzer, specify it in the "analyzers" section of an index at design time, and then reference it on searchable, Edm.String fields using either the "analyzer" property, or the "indexAnalyzer" and "searchAnalyzer" pair.

An analyzer definition includes a name, type, one or more character filters, a maximum of one tokenizer, and one or more token filters for post-tokenization processing. Character filters are applied before tokenization. Token filters and character filters are applied from left to right.

- Names in a custom analyzer must be unique and can't be the same as any of the built-in analyzers, tokenizers, token filters, or characters filters. It must only contain letters, digits, spaces, dashes or underscores, can only start and end with alphanumeric characters, and is limited to 128 characters.
- Type must be #Microsoft.Azure.Search.CustomAnalyzer.
- "charFilters" can be one or more filters from Character Filters, processed before tokenization, in the order provided. Some character filters have options, which can be set in a "charFilter section. Character filters are optional.
- "tokenizer" is exactly one Tokenizer. A value is required. If you need more than one tokenizer, you can create multiple custom analyzers and assign them on a field-by-field basis in your index schema.
- "tokenFilters" can be one or more filters from Token Filters, processed after tokenization, in the order provided. For token filters that have options, add a "tokenFilter" section to specify the configuration. Token filters are optional.

Analyzers must not produce tokens longer than 300 characters, or indexing will fail. To trim long token or to exclude them, use the **TruncateTokenFilter** and the

LengthTokenFilter respectively. See Token filters for reference.

JSON

```
"analyzers":(optional)[
   {
      "name":"name of analyzer",
      "@odata.type":"#Microsoft.Azure.Search.CustomAnalyzer",
      "charFilters":[
         "char_filter_name_1",
         "char_filter_name_2"
      ],
      "tokenizer":"tokenizer_name",
      "tokenFilters":[
         "token_filter_name_1",
         "token_filter_name_2"
      ]
   },
   {
      "name":"name of analyzer",
      "@odata.type":"#analyzer_type",
      "option1":value1,
      "option2":value2,
      ...
   }
],
"charFilters":(optional)[
   {
      "name":"char_filter_name",
      "@odata.type":"#char_filter_type",
      "option1":value1,
      "option2":value2,
      ...
   }
],
"tokenizers":(optional)[
   {
      "name":"tokenizer_name",
      "@odata.type":"#tokenizer_type",
      "option1":value1,
      "option2":value2,
      ...
   }
],
"tokenFilters":(optional)[
   {
      "name":"token_filter_name",
      "@odata.type":"#token_filter_type",
      "option1":value1,
      "option2":value2,
      ...
   }
]
```

Within an index definition, you can place this section anywhere in the body of a create index request but usually it goes at the end:

JSON

```
{
  "name": "name_of_index",
  "fields": [ ],
  "suggesters": [ ],
  "scoringProfiles": [ ],
  "defaultScoringProfile": (optional) "...",
  "corsOptions": (optional) { },
  "analyzers":(optional)[ ],
  "charFilters":(optional)[ ],
  "tokenizers":(optional)[ ],
  "tokenFilters":(optional)[ ]
}
```

The analyzer definition is a part of the larger index. Definitions for char filters, tokenizers, and token filters are added to the index only if you're setting custom options. To use an existing filter or tokenizer as-is, specify it by name in the analyzer definition. For more information, see Create Index (REST). For more examples, see Add analyzers in Azure Cognitive Search.

Test custom analyzers

You can use the Test Analyzer (REST) to see how an analyzer breaks given text into tokens.

Request

HTTP

```
  POST https://[search service name].search.windows.net/indexes/[index
name]/analyze?api-version=[api-version]
    Content-Type: application/json
    api-key: [admin key]

  {
     "analyzer":"my_analyzer",
     "text": "Vis-à-vis means Opposite"
  }
```

Response

HTTP

```
{
  "tokens": [
    {
      "token": "vis_a_vis",
      "startOffset": 0,
      "endOffset": 9,
      "position": 0
    },
    {
      "token": "vis_à_vis",
      "startOffset": 0,
      "endOffset": 9,
      "position": 0
    },
    {
      "token": "means",
      "startOffset": 10,
      "endOffset": 15,
      "position": 1
    },
    {
      "token": "opposite",
      "startOffset": 16,
      "endOffset": 24,
      "position": 2
    }
  ]
}
```

Update custom analyzers

Once an analyzer, a tokenizer, a token filter, or a character filter is defined, it can't be modified. New ones can be added to an existing index only if the `allowIndexDowntime` flag is set to true in the index update request:

HTTP

```
PUT https://[search service name].search.windows.net/indexes/[index name]?
api-version=[api-version]&allowIndexDowntime=true
```

This operation takes your index offline for at least a few seconds, causing your indexing and query requests to fail. Performance and write availability of the index can be impaired for several minutes after the index is updated, or longer for very large indexes, but these effects are temporary and eventually resolve on their own.

Built-in analyzers

If you want to use a built-in analyzer with custom options, creating a custom analyzer is the mechanism by which you specify those options. In contrast, to use a built-in analyzer as-is, you simply need to reference it by name in the field definition.

analyzer_name	analyzer_type [1]	Description and Options
keyword	(type applies only when options are available)	Treats the entire content of a field as a single token. This is useful for data like zip codes, IDs, and some product names.
pattern	PatternAnalyzer	Flexibly separates text into terms via a regular expression pattern. **Options** lowercase (type: bool) - Determines whether terms are lowercased. The default is true. pattern (type: string) - A regular expression pattern to match token separators. The default is `\W+`, which matches non-word characters. flags (type: string) - Regular expression flags. The default is an empty string. Allowed values: CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, LITERAL, MULTILINE, UNICODE_CASE, UNIX_LINES stopwords (type: string array) - A list of stopwords. The default is an empty list.
simple	(type applies only when options are available)	Divides text at non-letters and converts them to lower case.
standard (Also referred to as standard.lucene)	StandardAnalyzer	Standard Lucene analyzer, composed of the standard tokenizer, lowercase filter, and stop filter. **Options** maxTokenLength (type: int) - The maximum token length. The default is 255. Tokens longer than the maximum length are split. Maximum token length that can be used is 300 characters. stopwords (type: string array) - A list of stopwords. The default is an empty list.

analyzer_name	analyzer_type [1]	Description and Options
standardasciifolding.lucene	(type applies only when options are available)	Standard analyzer with Ascii folding filter.
stop	StopAnalyzer	Divides text at non-letters, applies the lowercase and stopword token filters. **Options** stopwords (type: string array) - A list of stopwords. The default is a predefined list for English.
whitespace	(type applies only when options are available)	An analyzer that uses the whitespace tokenizer. Tokens that are longer than 255 characters are split.

[1] Analyzer Types are always prefixed in code with "#Microsoft.Azure.Search" such that "PatternAnalyzer" would actually be specified as "#Microsoft.Azure.Search.PatternAnalyzer". We removed the prefix for brevity, but the prefix is required in your code.

The analyzer_type is only provided for analyzers that can be customized. If there are no options, as is the case with the keyword analyzer, there's no associated #Microsoft.Azure.Search type.

Character filters

Character filters add processing before a string reaches the tokenizer.

Cognitive Search supports character filters in the following list. More information about each one can be found in the Lucene API reference.

char_filter_name	char_filter_type [1]	Description and Options
html_strip	(type applies only when options are available)	A char filter that attempts to strip out HTML constructs.

char_filter_name	char_filter_type [1]	Description and Options
mapping	MappingCharFilter	A char filter that applies mappings defined with the mappings option. Matching is greedy (longest pattern matching at a given point wins). Replacement is allowed to be the empty string. **Options** mappings (type: string array) - A list of mappings of the following format: "a=>b" (all occurrences of the character "a" are replaced with character "b"). Required.
pattern_replace	PatternReplaceCharFilter	A char filter that replaces characters in the input string. It uses a regular expression to identify character sequences to preserve and a replacement pattern to identify characters to replace. For example, input text = "aa bb aa bb", pattern=" (aa)\\s+(bb)" replacement="$1#$2", result = "aa#bb aa#bb". **Options** pattern (type: string) - Required. replacement (type: string) - Required.

[1] Char Filter Types are always prefixed in code with "#Microsoft.Azure.Search" such that "MappingCharFilter" would actually be specified as "#Microsoft.Azure.Search.MappingCharFilter. We removed the prefix to reduce the width of the table, but please remember to include it in your code. Notice that char_filter_type is only provided for filters that can be customized. If there are no options, as is the case with html_strip, there's no associated #Microsoft.Azure.Search type.

Tokenizers

A tokenizer divides continuous text into a sequence of tokens, such as breaking a sentence into words, or a word into root forms.

Cognitive Search supports tokenizers in the following list. More information about each one can be found in the Lucene API reference.

tokenizer_name	tokenizer_type [1]	Description and Options

tokenizer_name	tokenizer_type [1]	Description and Options
classic	ClassicTokenizer	Grammar based tokenizer that is suitable for processing most European-language documents. **Options** maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.
edgeNGram	EdgeNGramTokenizer	Tokenizes the input from an edge into n-grams of given size(s). **Options** minGram (type: int) - Default: 1, maximum: 300. maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram. tokenChars (type: string array) - Character classes to keep in the tokens. Allowed values: "letter", "digit", "whitespace", "punctuation", "symbol". Defaults to an empty array - keeps all characters.

tokenizer_name	tokenizer_type [1]	Description and Options
keyword_v2	KeywordTokenizerV2	Emits the entire input as a single token. **Options** maxTokenLength (type: int) - The maximum token length. Default: 256, maximum: 300. Tokens longer than the maximum length are split.
letter	(type applies only when options are available)	Divides text at non-letters. Tokens that are longer than 255 characters are split.
lowercase	(type applies only when options are available)	Divides text at non-letters and converts them to lower case. Tokens that are longer than 255 characters are split.
microsoft_language_tokenizer	MicrosoftLanguageTokenizer	Divides text using language-specific rules. **Options** maxTokenLength (type: int) - The maximum token length, default: 255, maximum: 300. Tokens longer than the maximum length are split. Tokens longer than 300 characters are first split into tokens of length 300 and then each of those tokens is split based on the maxTokenLength set.

tokenizer_name	tokenizer_type [1]	Description and Options
		isSearchTokenizer (type: bool) - Set to true if used as the search tokenizer, set to false if used as the indexing tokenizer. language (type: string) - Language to use, default "english". Allowed values include: "bangla", "bulgarian", "catalan", "chineseSimplified", "chineseTraditional", "croatian", "czech", "danish", "dutch", "english", "french", "german", "greek", "gujarati", "hindi", "icelandic", "indonesian", "italian", "japanese", "kannada", "korean", "malay", "malayalam", "marathi", "norwegianBokmaal", "polish", "portuguese", "portugueseBrazilian", "punjabi", "romanian", "russian", "serbianCyrillic", "serbianLatin", "slovenian", "spanish", "swedish", "tamil", "telugu", "thai", "ukrainian", "urdu", "vietnamese"
microsoft_language_stemming_tokenizer	MicrosoftLanguageStemmingTokenizer	Divides text using language-specific rules and reduces words to their base forms. This tokenizer

tokenizer_name	tokenizer_type [1]	Description and Options
		performs lemmatization. **Options** maxTokenLength (type: int) - The maximum token length, default: 255, maximum: 300. Tokens longer than the maximum length are split. Tokens longer than 300 characters are first split into tokens of length 300 and then each of those tokens is split based on the maxTokenLength set. isSearchTokenizer (type: bool) - Set to true if used as the search tokenizer, set to false if used as the indexing tokenizer. language (type: string) - Language to use, default "english". Allowed values include: "arabic", "bangla", "bulgarian", "catalan", "croatian", "czech", "danish", "dutch", "english", "estonian", "finnish", "french", "german", "greek", "gujarati", "hebrew", "hindi", "hungarian", "icelandic", "indonesian", "italian", "kannada", "latvian", "lithuanian", "malay", "malayalam",

tokenizer_name	tokenizer_type [1]	Description and Options
		"marathi", "norwegianBokmaal", "polish", "portuguese", "portugueseBrazilian", "punjabi", "romanian", "russian", "serbianCyrillic", "serbianLatin", "slovak", "slovenian", "spanish", "swedish", "tamil", "telugu", "turkish", "ukrainian", "urdu"
nGram	NGramTokenizer	Tokenizes the input into n-grams of the given size(s). **Options** minGram (type: int) - Default: 1, maximum: 300. maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram. tokenChars (type: string array) - Character classes to keep in the tokens. Allowed values: "letter", "digit", "whitespace", "punctuation", "symbol". Defaults to an empty array - keeps all characters.

tokenizer_name	tokenizer_type [1]	Description and Options
path_hierarchy_v2	PathHierarchyTokenizerV2	Tokenizer for path-like hierarchies. **Options** delimiter (type: string) - Default: '/. replacement (type: string) - If set, replaces the delimiter character. Default same as the value of delimiter. maxTokenLength (type: int) - The maximum token length. Default: 300, maximum: 300. Paths longer than maxTokenLength are ignored. reverse (type: bool) - If true, generates token in reverse order. Default: false. skip (type: bool) - Initial tokens to skip. The default is 0.

tokenizer_name	tokenizer_type [1]	Description and Options
pattern	PatternTokenizer	This tokenizer uses regex pattern matching to construct distinct tokens. **Options** pattern (type: string) - Regular expression pattern to match token separators. The default is `\W+`, which matches non-word characters. flags (type: string) - Regular expression flags. The default is an empty string. Allowed values: CANON_EQ, CASE_INSENSITIVE, COMMENTS, DOTALL, LITERAL, MULTILINE, UNICODE_CASE, UNIX_LINES group (type: int) - Which group to extract into tokens. The default is -1 (split).

tokenizer_name	tokenizer_type [1]	Description and Options
standard_v2	StandardTokenizerV2	Breaks text following the Unicode Text Segmentation rules. **Options** maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.
uax_url_email	UaxUrlEmailTokenizer	Tokenizes urls and emails as one token. **Options** maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.
whitespace	(type applies only when options are available)	Divides text at whitespace. Tokens that are longer than 255 characters are split.

[1] Tokenizer Types are always prefixed in code with "#Microsoft.Azure.Search" such that "ClassicTokenizer" would actually be specified as "#Microsoft.Azure.Search.ClassicTokenizer". We removed the prefix to reduce the width of the table, but please remember to include it in your code. Notice that tokenizer_type is only provided for tokenizers that can be customized. If there are no options, as is the case with the letter tokenizer, there's no associated #Microsoft.Azure.Search type.

Token filters

A token filter is used to filter out or modify the tokens generated by a tokenizer. For example, you can specify a lowercase filter that converts all characters to lowercase. You can have multiple token filters in a custom analyzer. Token filters run in the order in which they're listed.

In the table below, the token filters that are implemented using Apache Lucene are linked to the Lucene API documentation.

token_filter_name	token_filter_type [1]	Description and Options
arabic_normalization	(type applies only when options are available)	A token filter that applies the Arabic normalizer to normalize the orthography.
apostrophe	(type applies only when options are available)	Strips all characters after an apostrophe (including the apostrophe itself).
asciifolding	AsciiFoldingTokenFilter	Converts alphabetic, numeric, and symbolic Unicode characters which aren't in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists. **Options** preserveOriginal (type: bool) - If true, the original token is kept. The default is false.

token_filter_name	token_filter_type [1]	Description and Options
cjk_bigram	CjkBigramTokenFilter	Forms bigrams of CJK terms that are generated from StandardTokenizer. **Options** ignoreScripts (type: string array) - Scripts to ignore. Allowed values include: "han", "hiragana", "katakana", "hangul". The default is an empty list. outputUnigrams (type: bool) - Set to true if you always want to output both unigrams and bigrams. The default is false.
cjk_width	(type applies only when options are available)	Normalizes CJK width differences. Folds full width ASCII variants into the equivalent basic latin and half-width Katakana variants into the equivalent kana.
classic	(type applies only when options are available)	Removes the English possessives, and dots from acronyms.

token_filter_name	token_filter_type [1]	Description and Options
common_grams	CommonGramTokenFilter	Construct bigrams for frequently occurring terms while indexing. Single terms are still indexed too, with bigrams overlaid. **Options** commonWords (type: string array) - The set of common words. The default is an empty list. Required. ignoreCase (type: bool) - If true, matching is case insensitive. The default is false. queryMode (type: bool) - Generates bigrams then removes common words and single terms followed by a common word. The default is false.

token_filter_name	token_filter_type [1]	Description and Options
dictionary_decompounder	DictionaryDecompounderTokenFilter	Decomposes compound words found in many Germanic languages. **Options** wordList (type: string array) - The list of words to match against. The default is an empty list. Required. minWordSize (type: int) - Only words longer than this get processed. The default is 5. minSubwordSize (type: int) - Only subwords longer than this are outputted. The default is 2. maxSubwordSize (type: int) - Only subwords shorter than this are outputted. The default is 15. onlyLongestMatch (type: bool) - Add only the longest matching subword to output. The default is false.

token_filter_name	token_filter_type [1]	Description and Options
edgeNGram_v2	EdgeNGramTokenFilterV2	Generates n-grams of the given size(s) from starting from the front or the back of an input token. **Options** minGram (type: int) - Default: 1, maximum: 300. maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram. side (type: string) - Specifies which side of the input the n-gram should be generated from. Allowed values: "front", "back"
elision	ElisionTokenFilter	Removes elisions. For example, "l'avion" (the plane) is converted to "avion" (plane). **Options** articles (type: string array) - A set of articles to remove. The default is an empty list. If there's no list of articles set, by default all French articles are removed.
german_normalization	(type applies only when options are available)	Normalizes German characters according to the heuristics of the German2 snowball algorithm .
hindi_normalization	(type applies only when options are available)	Normalizes text in Hindi to remove some differences in spelling variations.
indic_normalization	IndicNormalizationTokenFilter	Normalizes the Unicode representation of text in Indian languages.

token_filter_name	token_filter_type [1]	Description and Options
keep	KeepTokenFilter	A token filter that only keeps tokens with text contained in specified list of words. **Options** keepWords (type: string array) - A list of words to keep. The default is an empty list. Required. keepWordsCase (type: bool) - If true, lower case all words first. The default is false.
keyword_marker	KeywordMarkerTokenFilter	Marks terms as keywords. **Options** keywords (type: string array) - A list of words to mark as keywords. The default is an empty list. Required. ignoreCase (type: bool) - If true, lower case all words first. The default is false.
keyword_repeat	(type applies only when options are available)	Emits each incoming token twice once as keyword and once as non-keyword.
kstem	(type applies only when options are available)	A high-performance kstem filter for English.
length	LengthTokenFilter	Removes words that are too long or too short. **Options** min (type: int) - The minimum number. Default: 0, maximum: 300. max (type: int) - The maximum number. Default: 300, maximum: 300.

token_filter_name	token_filter_type [1]	Description and Options
limit	Microsoft.Azure.Search.LimitTokenFilter	Limits the number of tokens while indexing. **Options** maxTokenCount (type: int) - Max number of tokens to produce. The default is 1. consumeAllTokens (type: bool) - Whether all tokens from the input must be consumed even if maxTokenCount is reached. The default is false.
lowercase	(type applies only when options are available)	Normalizes token text to lower case.
nGram_v2	NGramTokenFilterV2	Generates n-grams of the given size(s). **Options** minGram (type: int) - Default: 1, maximum: 300. maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram.

token_filter_name	token_filter_type [1]	Description and Options
pattern_capture	PatternCaptureTokenFilter	Uses Java regexes to emit multiple tokens, one for each capture group in one or more patterns. **Options** patterns (type: string array) - A list of patterns to match against each token. Required. preserveOriginal (type: bool) - Set to true to return the original token even if one of the patterns matches, default: true
pattern_replace	PatternReplaceTokenFilter	A token filter which applies a pattern to each token in the stream, replacing match occurrences with the specified replacement string. **Options** pattern (type: string) - Required. replacement (type: string) - Required.
persian_normalization	(type applies only when options are available)	Applies normalization for Persian.

token_filter_name	token_filter_type [1]	Description and Options
phonetic	PhoneticTokenFilter	Create tokens for phonetic matches. **Options** encoder (type: string) - Phonetic encoder to use. Allowed values include: "metaphone", "doubleMetaphone", "soundex", "refinedSoundex", "caverphone1", "caverphone2", "cologne", "nysiis", "koelnerPhonetik", "haasePhonetik", "beiderMorse". Default: "metaphone". Default is metaphone. See encoder for more information. replace (type: bool) - True if encoded tokens should replace original tokens, false if they should be added as synonyms. The default is true.
porter_stem	(type applies only when options are available)	Transforms the token stream as per the Porter stemming algorithm.
reverse	(type applies only when options are available)	Reverses the token string.
scandinavian_normalization	(type applies only when options are available)	Normalizes use of the interchangeable Scandinavian characters.
scandinavian_folding	(type applies only when options are available)	Folds Scandinavian characters åÅäæÄÆ->a and öÖøØ->o. It also discriminates against use of double vowels aa, ae, ao, oe and oo, leaving just the first one.

token_filter_name	token_filter_type [1]	Description and Options
shingle	ShingleTokenFilter	Creates combinations of tokens as a single token. **Options** maxShingleSize (type: int) - Defaults to 2. minShingleSize (type: int) - Defaults to 2. outputUnigrams (type: bool) - if true, the output stream contains the input tokens (unigrams) as well as shingles. The default is true. outputUnigramsIfNoShingles (type: bool) - If true, override the behavior of outputUnigrams==false for those times when no shingles are available. The default is false. tokenSeparator (type: string) - The string to use when joining adjacent tokens to form a shingle. The default is " ". filterToken (type: string) - The string to insert for each position for which there is no token. The default is "_".

token_filter_name	token_filter_type [1]	Description and Options
snowball	SnowballTokenFilter	Snowball Token Filter. **Options** language (type: string) - Allowed values include: "armenian", "basque", "catalan", "danish", "dutch", "english", "finnish", "french", "german", "german2", "hungarian", "italian", "kp", "lovins", "norwegian", "porter", "portuguese", "romanian", "russian", "spanish", "swedish", "turkish"
sorani_normalization	SoraniNormalizationTokenFilter	Normalizes the Unicode representation of Sorani text. **Options** None.
stemmer	StemmerTokenFilter	Language-specific stemming filter. **Options** language (type: string) - Allowed values include: - "arabic" - "armenian" - "basque" - "brazilian" - "bulgarian" - "catalan" - "czech" - "danish" - "dutch" - "dutchKp" - "english" - "lightEnglish" - "minimalEnglish" - "possessiveEnglish" - "porter2" - "lovins" - "finnish"

token_filter_name	token_filter_type [1]	Description and Options
		- "lightFinnish" - "french" - "lightFrench" - "minimalFrench" - "galician" - "minimalGalician" - "german" - "german2" - "lightGerman" - "minimalGerman" - "greek" - "hindi" - "hungarian" - "lightHungarian" - "indonesian" - "irish" - "italian" - "lightItalian" - "sorani" - "latvian" - "norwegian" - "lightNorwegian" - "minimalNorwegian" - "lightNynorsk" - "minimalNynorsk" - "portuguese" - "lightPortuguese" - "minimalPortuguese" - "portugueseRslp" - "romanian" - "russian" - "lightRussian" - "spanish" - "lightSpanish" - "swedish" - "lightSwedish" - "turkish"

token_filter_name	token_filter_type [1]	Description and Options
stemmer_override	StemmerOverrideTokenFilter	Any dictionary-Stemmed terms are marked as keywords, which prevents stemming down the chain. Must be placed before any stemming filters. **Options** rules (type: string array) - Stemming rules in the following format "word => stem" for example "ran => run". The default is an empty list. Required.

token_filter_name	token_filter_type [1]	Description and Options
stopwords	StopwordsTokenFilter	Removes stop words from a token stream. By default, the filter uses a predefined stop word list for English. **Options** stopwords (type: string array) - A list of stopwords. Can't be specified if a stopwordsList is specified. stopwordsList (type: string) - A predefined list of stopwords. Can't be specified if "stopwords" is specified. Allowed values include:"arabic", "armenian", "basque", "brazilian", "bulgarian", "catalan", "czech", "danish", "dutch", "english", "finnish", "french", "galician", "german", "greek", "hindi", "hungarian", "indonesian", "irish", "italian", "latvian", "norwegian", "persian", "portuguese", "romanian", "russian", "sorani", "spanish", "swedish", "thai", "turkish", default: "english". Can't be specified if "stopwords" is specified. ignoreCase (type: bool) - If true, all words are lower cased first. The default is false. removeTrailing (type: bool) - If true, ignore the last search term if it's a stop word. The default is true.

token_filter_name	token_filter_type [1]	Description and Options
synonym	SynonymTokenFilter	Matches single or multi word synonyms in a token stream. **Options** synonyms (type: string array) - Required. List of synonyms in one of the following two formats: -incredible, unbelievable, fabulous => amazing - all terms on the left side of => symbol are replaced with all terms on its right side. -incredible, unbelievable, fabulous, amazing - A comma-separated list of equivalent words. Set the expand option to change how this list is interpreted. ignoreCase (type: bool) - Case-folds input for matching. The default is false. expand (type: bool) - If true, all words in the list of synonyms (if => notation is not used) map to one another. The following list: incredible, unbelievable, fabulous, amazing is equivalent to: incredible, unbelievable, fabulous, amazing => incredible, unbelievable, fabulous, amazing - If false, the following list: incredible, unbelievable, fabulous, amazing are equivalent to: incredible, unbelievable, fabulous, amazing => incredible.

token_filter_name	token_filter_type [1]	Description and Options
trim	(type applies only when options are available)	Trims leading and trailing whitespace from tokens.
truncate	TruncateTokenFilter	Truncates the terms into a specific length. **Options** length (type: int) - Default: 300, maximum: 300. Required.
unique	UniqueTokenFilter	Filters out tokens with same text as the previous token. **Options** onlyOnSamePosition (type: bool) - If set, remove duplicates only at the same position. The default is true.
uppercase	(type applies only when options are available)	Normalizes token text to upper case.
word_delimiter	WordDelimiterTokenFilter	Splits words into subwords and performs optional transformations on subword groups. **Options** generateWordParts (type: bool) - Causes parts of words to be generated, for example "AzureSearch" becomes "Azure" "Search". The default is true. generateNumberParts (type: bool) - Causes number subwords to be generated. The default is true. catenateWords (type: bool) - Causes maximum runs of word parts to be catenated, for example "Azure-Search"

token_filter_name	token_filter_type [1]	Description and Options
		becomes "AzureSearch". The default is false. catenateNumbers (type: bool) - Causes maximum runs of number parts to be catenated, for example "1-2" becomes "12". The default is false. catenateAll (type: bool) - Causes all subword parts to be catenated, e.g "Azure-Search-1" becomes "AzureSearch1". The default is false. splitOnCaseChange (type: bool) - If true, splits words on caseChange, for example "AzureSearch" becomes "Azure" "Search". The default is true. preserveOriginal - Causes original words to be preserved and added to the subword list. The default is false. splitOnNumerics (type: bool) - If true, splits on numbers, for example "Azure1Search" becomes "Azure" "1" "Search". The default is true. stemEnglishPossessive (type: bool) - Causes trailing "'s" to be removed for each subword. The default is true. protectedWords (type: string array) - Tokens to protect from being delimited. The default is an empty list.

[1] Token Filter Types are always prefixed in code with "#Microsoft.Azure.Search" such that "ArabicNormalizationTokenFilter" would actually be specified as

"#Microsoft.Azure.Search.ArabicNormalizationTokenFilter". We removed the prefix to reduce the width of the table, but please remember to include it in your code.

See also

- Azure Cognitive Search REST APIs
- Analyzers in Azure Cognitive Search (Examples)
- Create Index (REST)

Creating indexers in Azure Cognitive Search

Article • 03/24/2023 • 11 minutes to read

An indexer is a named object on a search service that automates an indexing workload by connecting to an external data source, retrieving and processing data, and then passing the data on to the search engine for indexing. Using indexers significantly reduces the quantity and complexity of the code you need to write.

Indexers support two workflows:

- Text-based indexing, extracting strings and metadata from textual content for full text search scenarios.
- Skills-based indexing, using built-in or custom skills that add integrated machine learning for analysis over images and large undifferentiated content, extracting or inferring text and structure. Skill-based indexing enables search over content that isn't otherwise easily full text searchable. To learn more, see AI enrichment in Cognitive Search.

This article focuses on the basic steps of creating an indexer. Depending on the data source and your workflow, more configuration might be necessary.

Indexer definitions

When you create an indexer, the definition will adhere to one of two patterns: text-based indexing or AI enrichment with skills. The patterns are the same except that skills-based indexing has more definitions.

Indexer definition for text-based indexing

Text-based indexing for full text search is the primary use case for indexers, and for this workflow, an indexer will look like this example.

JSON

```
{
  "name": (required) String that uniquely identifies the indexer,
  "description": (optional),
  "dataSourceName": (required) String indicating which existing data source
to use,
  "targetIndexName": (required) String indicating which existing index to
```

```
use,
  "parameters": {
    "batchSize": null,
    "maxFailedItems": 0,
    "maxFailedItemsPerBatch": 0,
    "base64EncodeKeys": false,
    "configuration": {}
  },
  "fieldMappings": (optional) unless field discrepancies need resolution,
  "disabled": null,
  "schedule": null,
  "encryptionKey": null
}
```

Indexers have the following requirements:

- A "name" property that uniquely identifies the indexer in the indexer collection.
- A "dataSourceName" property that points to a data source object. It specifies a connection to external data.
- A "targetIndexName" property that points to the destination search index.

Other parameters are optional and modify run time behaviors, such as how many errors to accept before failing the entire job. The parameters above are available for all indexers and are documented in the REST API reference.

Data source-specific indexers for blobs, SQL, and Azure Cosmos DB provide extra "configuration" parameters for source-specific behaviors. For example, if the source is Blob Storage, you can set a parameter that filters on file extensions: `"parameters" : { "configuration" : { "indexedFileNameExtensions" : ".pdf,.docx" } }`. If the source is Azure SQL, you can set a query time out parameter.

Field mappings are used to explicitly map source-to-destination fields if there are discrepancies by name or type between a field in the data source and a field in the search index.

By default, an indexer runs immediately when you create it on the search service. If you don't want indexer execution, set "disabled" to true when creating the indexer.

You can also specify a schedule or set an encryption key for supplemental encryption of the indexer definition.

Indexer definition for skills-based indexing and AI enrichment

Indexers also drive AI enrichment. All of the above properties and parameters for apply, but the following extra properties are specific to AI enrichment: "skillSetName", "cache", "outputFieldMappings".

JSON

```
{
  "name": (required) String that uniquely identifies the indexer,
  "dataSourceName": (required) String, provides raw content that will be
enriched,
  "targetIndexName": (required) String, name of an existing index,
  "skillsetName" : (required for AI enrichment) String, name of an existing
skillset,
  "cache":  {
    "storageConnectionString" : (required if you enable the cache)
Connection string to a blob container,
    "enableReprocessing": true
    },
  "parameters": { },
  "fieldMappings": (optional) Maps fields in the underlying data source to
fields in an index,
  "outputFieldMappings" : (required) Maps skill outputs to fields in an
index,
}
```

AI enrichment is its own subject area and is out of scope for this article. For more information, start with AI enrichment, Skillsets in Azure Cognitive Search, Create a skillset, Map enrichment output fields, and Enable caching for AI enrichment.

Prerequisites

- Identify a supported data source that contains the content you want to ingest.
- Create an indexer data source that sets up a connection to external data.
- Create a search index that can accept incoming data.
- Be under the maximum limits for your service tier. The Free tier allows three objects of each type and 1-3 minutes of indexer processing or 3-10 if there's a skillset.

Prepare external data

Indexers work with data sets. When you run an indexer, it connects to your data source, retrieves the data from the container or folder, optionally serializes it into JSON before

passing it to the search engine for indexing. This section describes the requirements of incoming data for text-based indexing.

Source data	Tasks
JSON documents	Make sure the structure or shape of incoming data corresponds to the schema of your search index. Most search indexes are fairly flat, where the fields collection consists of fields at the same level. However, hierarchical or nested structures are possible through complex fields and collections.
Relational	You'll need to provide it as a flattened row set, where each row becomes a full or partial search document in the index. To flatten relational data into a row set, you should create a SQL view, or build a query that returns parent and child records in the same row. For example, the built-in hotels sample dataset is an SQL database that has 50 records (one for each hotel), linked to room records in a related table. The query that flattens the collective data into a row set embeds all of the room information in JSON documents in each hotel record. The embedded room information is a generated by a query that uses a **FOR JSON AUTO** clause. You can learn more about this technique in define a query that returns embedded JSON. This is just one example; you can find other approaches that will produce the same result.
Files	An indexer generally creates one search document for each file, where the search document consists of fields for content and metadata. Depending on the file type, the indexer can sometimes parse one file into multiple search documents. For example, in a CSV file, each row can become a standalone search document.

Remember that you'll only need to pull in searchable and filterable data:

- Searchable data is text.
- Filterable data is alphanumeric.

Cognitive Search can't search over binary data in any format, although it can extract and infer text descriptions of image files (see AI enrichment) to create searchable content. Likewise, large text can be broken down and analyzed by natural language models to find structure or relevant information, generating new content that you can add to a search document.

Given that indexers don't fix data problems, other forms of data cleansing or manipulation might be needed. For more information, you should refer to the product documentation of your Azure database product.

Prepare a data source

Indexers require a data source that specifies the type, container, and connection.

1. Make sure you're using a supported data source type.
2. Create a data source definition. The following list is a few of the more frequently used data sources:
   - Azure Blob Storage
   - Azure Cosmos DB
   - Azure SQL Database
3. If the data source is a database, such as Azure SQL or Cosmos DB, enable change tracking. The above links for the various data sources explain which change tracking methods are supported by indexers.

Prepare an index

Indexers also require a search index. Recall that indexers pass data off to the search engine for indexing. Just as indexers have properties that determine execution behavior, an index schema has properties that profoundly affect how strings are indexed (only strings are analyzed and tokenized).

1. Start with Create a search index.
2. Set up the fields collection and field attributes.

   Fields are the only receptors of external content. Depending on how the fields are attributed in the schema, the values for each field will be analyzed, tokenized, or stored as verbatim strings for filters, fuzzy search, and typeahead queries.

   Indexers can automatically map source fields to target index fields when the names and types are equivalent. If a field can't be implicitly mapped, remember that you can define an explicit field mapping that tells the indexer how to route the content.
3. Review the analyzer assignments on each field. Analyzers can transform strings. As such, indexed strings might be different from what you passed in. You can evaluate the effects of analyzers using Analyze Text (REST). For more information about analyzers, see Analyzers for text processing.

During indexing, an indexer only checks field names and types. There's no validation step that ensures incoming content is correct for the corresponding search field in the index.

Create an indexer

When you're ready to create an indexer on a remote search service, you'll need a search client. A search client can be the Azure portal, Postman or another REST client, or code that instantiates an indexer client. We recommend the Azure portal or REST APIs for early development and proof-of-concept testing.

Azure portal

1. Sign in to Azure portal.
2. On the search service Overview page, choose from two options:
   - **Import data wizard**. The wizard is unique in that it creates all of the required elements. Other approaches require a predefined data source and index.
   - **New Indexer**, a visual editor for specifying an indexer definition.

   The following screenshot shows where you can find these features in the portal.

Run the indexer

By default, an indexer runs immediately when you create it on the search service. You can override this behavior by setting "disabled" to true in the indexer definition. Indexer execution is the moment of truth where you'll find out if there are problems with connections, field mappings, or skillset construction.

There are several ways to run an indexer:

- Run on indexer creation or update (default).
- Run on demand when there are no changes to the definition, or precede with reset for full indexing. For more information, see Run or reset indexers.
- Schedule indexer processing to invoke execution at regular intervals.

Scheduled execution is usually implemented when you have a need for incremental indexing so that you can pick up the latest changes. As such, scheduling has a dependency on change detection.

Check results

Monitor indexer status to check for status. Successful execution can still include warning and notifications. Be sure to check both successful and failed status notifications for details about the job.

For content verification, run queries on the populated index that return entire documents or selected fields.

Change detection and internal state

If your data source supports change detection, an indexer can detect underlying changes in the data and process just the new or updated documents on each indexer run, leaving unchanged content as-is. If indexer execution history says that a run was successful with `0/0` documents processed, it means that the indexer didn't find any new or changed rows or blobs in the underlying data source.

Change detection logic is built into the data platforms. How an indexer supports change detection varies by data source:

- Azure Storage has built-in change detection, which means an indexer can recognize new and updated documents automatically. Blob Storage, Azure Table Storage, and Azure Data Lake Storage Gen2 stamp each blob or row update with a date and time. An indexer automatically uses this information to determine which documents to update in the index. For more information about deletion detection, see Delete detection using indexers for Azure Storage in Azure Cognitive Search.
- Cloud database technologies provide optional change detection features in their platforms. For these data sources, change detection isn't automatic. You'll need to specify in the data source definition which change detection policy is used:

- Azure SQL (change detection)
- Azure DB for MySQL (change detection)
- Azure Cosmos DB for NoSQL (change detection)
- Azure Cosmos DB for MongoDB (change detection)
- Azure CosmosDB for Apache Gremlin (change detection)

Indexers keep track of the last document it processed from the data source through an internal "high water mark". The marker is never exposed in the API, but internally the indexer keeps track of where it stopped. When indexing resumes, either through a scheduled run or an on-demand invocation, the indexer references the high water mark so that it can pick up where it left off.

If you need to clear the high water mark to reindex in full, you can use Reset Indexer. For more selective reindexing, use Reset Skills or Reset Documents. Through the reset APIs, you can clear internal state, and also flush the cache if you enabled incremental enrichment. For more background and comparison of each reset option, see Run or reset indexers, skills, and documents.

Next steps

- Index data from Azure Blob Storage
- Index data from Azure SQL database
- Index data from Azure Data Lake Storage Gen2
- Index data from Azure Table Storage
- Index data from Azure Cosmos DB

Run or reset indexers, skills, or documents

Article • 12/08/2022 • 10 minutes to read

In Azure Cognitive Search, there are several ways to run an indexer:

- Run when creating or updating an indexer, assuming it's not created in "disabled" mode.
- Run on a schedule to invoke execution at regular intervals.
- Run on demand, with or without a "reset".

This article explains how to run indexers on demand, with and without a reset. It also describes indexer execution, duration, and concurrency.

Indexer execution

You can run multiple indexers at one time, but each indexer itself is single-instance. Starting a new instance while the indexer is already in execution produces this error: `"Failed to run indexer "<indexer name>" error: "Another indexer invocation is currently in progress; concurrent invocations are not allowed."`

An indexer job runs in a managed execution environment. Currently, there are two environments. You can't control or configure which environment is used. Azure Cognitive Search determines the environment based on job composition and the ability of the service to move an indexer job onto a content processor (some security features block the multi-tenant environment).

Indexer execution environments include:

- A private execution environment that runs on search nodes, specific to your search service.

- A multi-tenant environment with content processors, managed and secured by Microsoft at no extra cost. This environment is used to offload computationally intensive processing, leaving service-specific resources available for routine operations. Whenever possible, most indexer jobs are executed in the multi-tenant environment.

Indexer limits vary for each environment:

Workload	Maximum duration	Maximum jobs	Execution environment
Private execution	24 hours	One indexer job per search unit [1].	Indexing doesn't run in the background. Instead, the search service will balance all indexing jobs against ongoing queries and object management actions (such as creating or updating indexes). When running indexers, you should expect to see some query latency if indexing volumes are large.
Multi-tenant	2 hours [2]	Indeterminate [3]	Because the content processing cluster is multi-tenant, nodes are added to meet demand. If you experience a delay in on-demand or scheduled execution, it's probably because the system is either adding nodes or waiting for one to become available.

[1] Search units can be flexible combinations of partitions and replicas, but indexer jobs aren't tied to one or the other. In other words, if you have 12 units, you can have 12 indexer jobs running concurrently in private execution, no matter how the search units are deployed.

[2] If more than two hours are needed to process all of the data, enable change detection and schedule the indexer to run at two hour intervals. See Indexing a large data set for more strategies.

[3] "Indeterminate" means that the limit isn't quantified by the number of jobs. Some workloads, such as skillset processing, can run in parallel which could result in many jobs even though only one indexer is involved. Although the environment doesn't impose constraints, indexer limits for your search service still apply.

Run without reset

A Run Indexer operation will detect and process only what it necessary to synchronize the search index with changes in the underlying data source. Incremental indexing starts by locating an internal high-water mark to find the last updated search document, which becomes the starting point for indexer execution over new and updated documents in the data source.

Change detection is essential for determining what's new or updated in the data source. Indexers use the change detection capabilities of the underlying data source to determine what's new or updated in the data source.

- Azure Storage has built-in change detection through its LastModified property.

- Other data sources, such as Azure SQL or Azure Cosmos DB, have to be configured for change detection before the indexer can read new and updated rows.

If the underlying content is unchanged, a run operation has no effect. In this case, indexer execution history will indicate `0\0` documents processed.

You'll need to reset the indexer, as explained in the next section, to reprocess in full.

Resetting indexers

After the initial run, an indexer keeps track of which search documents have been indexed through an internal *high-water mark*. The marker is never exposed, but internally the indexer knows where it last stopped.

If you need to rebuild all or part of an index, you can clear the indexer's high-water mark through a reset. Reset APIs are available at decreasing levels in the object hierarchy:

- Reset Indexers clears the high-water mark and performs a full reindex of all documents
- Reset Documents (preview) reindexes a specific document or list of documents
- Reset Skills (preview) invokes skill processing for a specific skill

After reset, follow with a Run command to reprocess new and existing documents. Orphaned search documents having no counterpart in the data source can't be removed through reset/run. If you need to delete documents, see Add, Update or Delete Documents instead.

How to reset and run indexers

Reset clears the high-water mark. All documents in the search index will be flagged for full overwrite, without inline updates or merging into existing content. For indexers with a skillset and enrichment caching, resetting the index will also implicitly reset the skillset.

The actual work occurs when you follow a reset with a Run command:

- All new documents found the underlying source will be added to the search index.
- All documents that exist in both the data source and search index will be overwritten in the search index.
- Any enriched content created from skillsets will be rebuilt. The enrichment cache, if one is enabled, is refreshed.

As previously noted, reset is a passive operation: you must follow up a Run request to rebuild the index.

Reset/run operations apply to a search index or a knowledge store, to specific documents or projections, and to cached enrichments if a reset explicitly or implicitly includes skills.

Reset also applies to create and update operations. It will not trigger deletion or clean up of orphaned documents in the search index. For more information about deleting documents, see Add, Update or Delete Documents.

Once you reset an indexer, you can't undo the action.

Azure portal

1. Sign in to Azure portal and open the search service page.
2. On the **Overview** page, select the **Indexers** tab.
3. Select an indexer.
4. Select the **Reset** command, and then select **Yes** to confirm the action.
5. Refresh the page to show the status. You can select the item to view its details.
6. Select **Run** to start indexer processing, or wait for the next scheduled execution.

How to reset skills (preview)

For indexers that have skillsets, you can reset individual skills to force processing of just that skill and any downstream skills that depend on its output. The enrichment cache, if you enabled it, is also refreshed.

Reset Skills is currently REST-only, available through `api-version=2020-06-30-Preview` or later.

```HTTP
POST /skillsets/[skillset name]/resetskills?api-version=2020-06-30-Preview
{
    "skillNames" : [
        "#1",
        "#5",
        "#6"
    ]
}
```

You can specify individual skills, as indicated in the example above, but if any of those skills require output from unlisted skills (#2 through #4), unlisted skills will run unless the cache can provide the necessary information. In order for this to be true, cached enrichments for skills #2 through #4 must not have dependency on #1 (listed for reset).

If no skills are specified, the entire skillset is executed and if caching is enabled, the cache is also refreshed.

Remember to follow up with Run Indexer to invoke actual processing.

How to reset docs (preview)

The Reset Documents API accepts a list of document keys so that you can refresh specific documents. If specified, the reset parameters become the sole determinant of what gets processed, regardless of other changes in the underlying data. For example, if 20 blobs were added or updated since the last indexer run, but you only reset one document, only that document is processed.

On a per-document basis, all fields in that search document are refreshed with values from the data source. You can't pick and choose which fields to refresh.

If the document is enriched through a skillset and has cached data, the skillset is invoked for just the specified documents, and the cache is updated for the reprocessed documents.

When you're testing this API for the first time, the following APIs can help you validate and test the behaviors:

1. Call Get Indexer Status with API version `api-version=2020-06-30-Preview` or later, to check reset status and execution status. You can find information about the

reset request at the end of the status response.

2. Call Reset Documents with API version `api-version=2020-06-30-Preview` or later, to specify which documents to process.

   HTTP

   ```
   POST https://[service name].search.windows.net/indexers/[indexer name]/resetdocs?api-version=2020-06-30-Preview
   {
       "documentKeys" : [
           "1001",
           "4452"
       ]
   }
   ```

   - The document keys provided in the request are values from the search index, which can be different from the corresponding fields in the data source. If you're unsure of the key value, send a query to return the value.You can use `select` to return just the document key field.

   - For blobs that are parsed into multiple search documents (where parsingMode is set to jsonLines or jsonArrays, or delimitedText), the document key is generated by the indexer and might be unknown to you. In this scenario, a query for the document key to return the correct value.

3. Call Run Indexer (any API version) to process the documents you specified. Only those specific documents are indexed.

4. Call Run Indexer a second time to process from the last high-water mark.

5. Call Search Documents to check for updated values, and also to return document keys if you're unsure of the value. Use `"select": "<field names>"` if you want to limit which fields appear in the response.

Overwriting the document key list

Calling Reset Documents API multiple times with different keys appends the new keys to the list of document keys reset. Calling the API with the **`overwrite`** parameter set to true will overwrite the current list with the new one:

HTTP

```
POST https://[service name].search.windows.net/indexers/[indexer name]/resetdocs?api-version=2020-06-30-Preview
```

```
{
    "documentKeys" : [
        "200",
        "630"
    ],
    "overwrite": true
}
```

Check reset status "currentState"

To check reset status and to see which document keys are queued up for processing, following these steps.

1. Call Get Indexer Status with `api-version=06-30-2020-Preview` or later.

   The preview API will return the **`currentState`** section, found at the end of the response.

   JSON

   ```
   "currentState": {
       "mode": "indexingResetDocs",
       "allDocsInitialTrackingState": "
   {\"LastFullEnumerationStartTime\":\"2021-02-
   06T19:02:07.0323764+00:00\",\"LastAttemptedEnumerationStartTime\":\"202
   1-02-06T19:02:07.0323764+00:00\",\"NameHighWaterMark\":null}",
       "allDocsFinalTrackingState": "
   {\"LastFullEnumerationStartTime\":\"2021-02-
   06T19:02:07.0323764+00:00\",\"LastAttemptedEnumerationStartTime\":\"202
   1-02-06T19:02:07.0323764+00:00\",\"NameHighWaterMark\":null}",
       "resetDocsInitialTrackingState": null,
       "resetDocsFinalTrackingState": null,
       "resetDocumentKeys": [
           "200",
           "630"
       ]
   }
   ```

2. Check the "mode":

   For Reset Skills, "mode" should be set to **`indexingAllDocs`** (because potentially all documents are affected, in terms of the fields that are populated through AI enrichment).

   For Reset Documents, "mode" should be set to **`indexingResetDocs`**. The indexer retains this status until all the document keys provided in the reset documents call are processed, during which time no other indexer jobs will execute while the

operation is progressing. Finding all of the documents in the document keys list requires cracking each document to locate and match on the key, and this can take a while if the data set is large. If a blob container contains hundreds of blobs, and the docs you want to reset are at the end, the indexer won't find the matching blobs until all of the others have been checked first.

3. After the documents are reprocessed, run Get Indexer Status again. The indexer returns to the `indexingAllDocs` mode and will process any new or updated documents on the next run.

Next steps

Reset APIs are used to inform the scope of the next indexer run. For actual processing, you'll need to invoke an on-demand indexer run or allow a scheduled job to complete the work. After the run is finished, the indexer returns to normal processing, whether that is on a schedule or on-demand processing.

After you reset and rerun indexer jobs, you can monitor status from the search service, or obtain detailed information through resource logging.

- Indexer operations (REST)
- Monitor search indexer status
- Collect and analyze log data
- Schedule an indexer

Schedule an indexer in Azure Cognitive Search

Article • 12/08/2022 • 5 minutes to read

Indexers can be configured to run on a schedule when you set the "schedule" property. Some situations where indexer scheduling is useful include:

- Source data is changing over time, and you want the indexer to automatically process the difference.
- Source data is very large, and you need a recurring schedule to index all of the content.
- An index is populated from multiple sources, using multiple indexers, and you want to stagger the jobs to reduce conflicts.

When indexing can't complete within the typical 2-hour processing window, you can schedule the indexer to run on a 2-hour cadence to work through a large volume of data. As long as your data source supports change detection logic, indexers can automatically pick up where they left off on each run.

Once an indexer is on a schedule, it remains on the schedule until you clear the interval or start time, or set "disabled" to true. Leaving the indexer on a schedule when there's nothing to process won't impact system performance. Checking for changed content is a relatively fast operation.

Prerequisites

- A valid indexer configured with a data source and index.
- Change detection in the data source. Azure Storage and SharePoint have built-in change detection. Other data sources, such as Azure SQL and Azure Cosmos DB must be enabled manually.

Schedule definition

A schedule is part of the indexer definition. If the "schedule" property is omitted, the indexer will only run on demand. The property has two parts.

Property	Description

Property	Description
"interval"	(required) The amount of time between the start of two consecutive indexer executions. The smallest interval allowed is 5 minutes, and the longest is 1440 minutes (24 hours). It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). The pattern for this is: `P(nD)(T(nH)(nM))`. Examples: `PT15M` for every 15 minutes, `PT2H` for every two hours.
"startTime"	(optional) Start time is specified in coordinated universal time (UTC). If omitted, the current time is used. This time can be in the past, in which case the first execution is scheduled as if the indexer has been running continuously since the original start time.

The following example is a schedule that starts on January 1 at midnight and runs every two hours.

JSON

```
{
    "dataSourceName" : "hotels-ds",
    "targetIndexName" : "hotels-idx",
    "schedule" : { "interval" : "PT2H", "startTime" : "2022-01-01T00:00:00Z"
}
}
```

Configure a schedule

Schedules are specified in an indexer definition. To set up a schedule, you can use Azure portal, REST APIs, or an Azure SDK.

Azure portal

1. Sign in to Azure portal and open the search service page.
2. On the **Overview** page, select the **Indexers** tab.
3. Select an indexer.
4. Select **Settings**.
5. Scroll down to **Schedule**, and then choose Hourly, Daily, or Custom to set a specific date, time, or custom interval.

Scheduling behavior

For text-based indexing, the scheduler can kick off as many indexer jobs as the search service supports, which is determined by the number of search units. For example, if the service has three replicas and four partitions, you can generally have 12 indexer jobs in active execution, whether initiated on demand or on a schedule.

Skills-based indexers run in a different execution environment. For this reason, the number of service units has no bearing on the number of skills-based indexer jobs you can run. Multiple skills-based indexers can run in parallel, but doing so depends on node availability within the execution environment.

Although multiple indexers can run simultaneously, a given indexer is single instance. You can't run two copies of the same indexer concurrently. If an indexer happens to still be running when its next scheduled execution is set to start, the pending execution is postponed until the next scheduled occurrence, allowing the current job to finish.

Let’s consider an example to make this more concrete. Suppose we configure an indexer schedule with an interval of hourly and a start time of June 1, 2022 at 8:00:00 AM UTC. Here's what could happen when an indexer run takes longer than an hour:

- The first indexer execution starts at or around June 1, 2022 at 8:00 AM UTC. Assume this execution takes 20 minutes (or any amount of time that's less than 1 hour).
- The second execution starts at or around June 1, 2022 9:00 AM UTC. Suppose that this execution takes 70 minutes - more than an hour – and it will not complete until 10:10 AM UTC.
- The third execution is scheduled to start at 10:00 AM UTC, but at that time the previous execution is still running. This scheduled execution is then skipped. The next execution of the indexer won't start until 11:00 AM UTC.

> **Note**
>
> If an indexer is set to a certain schedule but repeatedly fails on the same document each time, the indexer will begin running on a less frequent interval (up to the maximum interval of at least once every 2 hours or 24 hours, depending on different implementation factors) until it successfully makes progress again. If you believe you have fixed whatever the underlying issue, you can **run the indexer manually**, and if indexing succeeds, the indexer will return to its regular schedule.

Next steps

For indexers that run on a schedule, you can monitor operations by retrieving status from the search service, or obtain detailed information by enabling resource logging.

- Monitor search indexer status
- Collect and analyze log data
- Index large data sets

Additional resources

🕮 Documentation

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Create Data Source (Azure Cognitive Search REST API)

Data source connection information used by an Azure Cognitive Search indexer when crawling external data sources.

Run or reset indexers - Azure Cognitive Search

Run indexers in full, or reset an indexer, skills, or individual documents to refresh all or part of a search index or knowledge store.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Add custom analyzers to string fields - Azure Cognitive Search

Configure text tokenizers and character filters to perform text analysis on strings during indexing and queries.

Show 5 more

Field mappings and transformations using Azure Cognitive Search indexers

Article • 09/15/2022 • 11 minutes to read

When an Azure Cognitive Search indexer loads a search index, it determines the data path through source-to-destination field mappings. Implicit field mappings are internal and occur when field names and data types are compatible between the source and destination.

If inputs and outputs don't match, you can define explicit *field mappings* to set up the data path, as described in this article. Field mappings can also be used to introduce light-weight data conversion, such as encoding or decoding, through mapping functions. If more processing is required, consider Azure Data Factory to bridge the gap.

Field mappings apply to:

- Physical data structures on both sides of the data stream (between a supported data source and a search index). If you're importing skill-enriched content that resides in memory, use outputFieldMappings instead.
- Search indexes only. If you're populating a knowledge store, use projections for data path configuration.
- Top-level search fields only, where the "targetFieldName" is either a simple field or a collection. A target field can't be a complex type.

> **Note**
>
> If you're working with complex data (nested or hierarchical structures), and you'd like to mirror that data structure in your search index, your search index must match the source structure exactly (same field names, levels, and types) so that the default mappings will work. Optionally, you might want just a few nodes in the complex structure. To get individual nodes, you can flatten incoming data into a string collection (see **outputFieldMappings** for this workaround).

Supported scenarios

Use-case	Description
Name discrepancy	Suppose your data source has a field named `_city`. Given that Azure Cognitive Search doesn't allow field names that start with an underscore, a field mapping lets you effectively map "_city" to "city". If your indexing requirements include retrieving content from multiple data sources, where field names vary among the sources, you could use a field mapping to clarify the path.
Type discrepancy	Supposed you want a source integer field to be of type `Edm.String` so that it's searchable in the search index. Because the types are different, you'll need to define a field mapping in order for the data path to succeed. Note that Cognitive Search has a smaller set of supported data types than many data sources. If you're importing SQL data, a field mapping allows you to map the SQL data type you want in a search index.
One-to-many data paths	You can populate multiple fields in the index with content from the same source field. For example, you might want to apply different analyzers to each field to support different use cases in your client app.
Encoding and decoding	You can apply mapping functions to support Base64 encoding or decoding of data during indexing.
Split strings or recast arrays into collections	You can apply mapping functions to split a string that includes a delimiter, or to send a JSON array to a search field of type `Collection(Edm.String)`.

Define a field mapping

Field mappings are added to the "fieldMappings" array of an indexer definition. A field mapping consists of three parts.

JSON

```
"fieldMappings": [
  {
    "sourceFieldName": "_city",
    "targetFieldName": "city",
    "mappingFunction": null
  }
],
```

Property	Description
sourceFieldName	Required. Represents a field in your data source.

Property	Description
targetFieldName	Optional. Represents a field in your search index. If omitted, the value of "sourceFieldName" is assumed for the target. Target fields must be top-level simple fields or collections. It can't be a complex type or collection. If you're handling a data type issue, a field's data type is specified in the index definition. The field mapping just needs to have the field's name.
mappingFunction	Optional. Consists of predefined functions that transform data.

Azure Cognitive Search uses case-insensitive comparison to resolve the field and function names in field mappings. This is convenient (you don't have to get all the casing right), but it means that your data source or index can't have fields that differ only by case.

> **Note**
>
> If no field mappings are present, indexers assume data source fields should be mapped to index fields with the same name. Adding a field mapping overrides the default field mappings for the source and target field. Some indexers, such as the **blob storage indexer**, add default field mappings for the index key field.

You can use the REST API or an Azure SDK to define field mappings.

REST APIs

Use Create Indexer (REST) or Update Indexer (REST), any API version.

This example handles a field name discrepancy.

JSON

```
PUT https://[service name].search.windows.net/indexers/myindexer?api-version=[api-version]
Content-Type: application/json
api-key: [admin key]
{
    "dataSourceName" : "mydatasource",
    "targetIndexName" : "myindex",
    "fieldMappings" : [ { "sourceFieldName" : "_city", "targetFieldName" : "city" } ]
}
```

This example maps a single source field to multiple target fields ("one-to-many" mappings). You can "fork" a field, copying the same source field content to two

different index fields that will be analyzed or attributed differently in the index.

JSON

```
"fieldMappings" : [
    { "sourceFieldName" : "text", "targetFieldName" :
"textStandardEnglishAnalyzer" },
    { "sourceFieldName" : "text", "targetFieldName" :
"textSoundexAnalyzer" }
]
```

Mapping functions and examples

A field mapping function transforms the contents of a field before it's stored in the index. The following mapping functions are currently supported:

- base64Encode
- base64Decode
- extractTokenAtPosition
- jsonArrayToStringCollection
- urlEncode
- urlDecode

base64Encode function

Performs *URL-safe* Base64 encoding of the input string. Assumes that the input is UTF-8 encoded.

Example: Base-encoding a document key

Only URL-safe characters can appear in an Azure Cognitive Search document key (so that you can address the document using the Lookup API). If the source field for your key contains URL-unsafe characters, such as `-` and `\`, use the `base64Encode` function to convert it at indexing time.

The following example specifies the base64Encode function on "metadata_storage_name" to handle unsupported characters.

HTTP

```
PUT /indexers?api-version=2020-06-30
{
```

```
  "dataSourceName" : "my-blob-datasource ",
  "targetIndexName" : "my-search-index",
  "fieldMappings" : [
    {
        "sourceFieldName" : "metadata_storage_name",
        "targetFieldName" : "key",
        "mappingFunction" : {
            "name" : "base64Encode",
            "parameters" : { "useHttpServerUtilityUrlTokenEncode" : false }
        }
    }
  ]
}
```

A document key (both before and after conversion) can't be longer than 1,024 characters. When you retrieve the encoded key at search time, use the `base64Decode` function to get the original key value, and use that to retrieve the source document.

Example: Make a base-encoded field "searchable"

There are times when you need to use an encoded version of a field like "metadata_storage_path" as the key, but also need an unencoded version for full text search. To support both scenarios, you can map "metadata_storage_path" to two fields: one for the key (encoded), and a second for a path field that we can assume is attributed as "searchable" in the index schema.

HTTP

```
PUT /indexers/blob-indexer?api-version=2020-06-30
{
    "dataSourceName" : " blob-datasource ",
    "targetIndexName" : "my-target-index",
    "schedule" : { "interval" : "PT2H" },
    "fieldMappings" : [
        { "sourceFieldName" : "metadata_storage_path", "targetFieldName" :
"key", "mappingFunction" : { "name" : "base64Encode" } },
        { "sourceFieldName" : "metadata_storage_path", "targetFieldName" :
"path" }
      ]
}
```

Example - preserve original values

The blob storage indexer automatically adds a field mapping from `metadata_storage_path`, the URI of the blob, to the index key field if no field mapping is specified. This value is Base64 encoded so it's safe to use as an Azure Cognitive Search

document key. The following example shows how to simultaneously map a *URL-safe* Base64 encoded version of `metadata_storage_path` to a `index_key` field and preserve the original value in a `metadata_storage_path` field:

JSON

```
"fieldMappings": [
  {
    "sourceFieldName": "metadata_storage_path",
    "targetFieldName": "metadata_storage_path"
  },
  {
    "sourceFieldName": "metadata_storage_path",
    "targetFieldName": "index_key",
    "mappingFunction": {
       "name": "base64Encode"
    }
  }
]
```

If you don't include a parameters property for your mapping function, it defaults to the value `{"useHttpServerUtilityUrlTokenEncode" : true}`.

Azure Cognitive Search supports two different Base64 encodings. You should use the same parameters when encoding and decoding the same field. For more information, see base64 encoding options to decide which parameters to use.

base64Decode function

Performs Base64 decoding of the input string. The input is assumed to be a *URL-safe* Base64-encoded string.

Example - decode blob metadata or URLs

Your source data might contain Base64-encoded strings, such as blob metadata strings or web URLs, that you want to make searchable as plain text. You can use the `base64Decode` function to turn the encoded data back into regular strings when populating your search index.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "Base64EncodedMetadata",
    "targetFieldName" : "SearchableMetadata",
    "mappingFunction" : {
```

```
      "name" : "base64Decode",
      "parameters" : { "useHttpServerUtilityUrlTokenDecode" : false }
    }
  }]
```

If you don't include a parameters property, it defaults to the value `{"useHttpServerUtilityUrlTokenEncode" : true}`.

Azure Cognitive Search supports two different Base64 encodings. You should use the same parameters when encoding and decoding the same field. For more information, see base64 encoding options to decide which parameters to use.

base64 encoding options

Azure Cognitive Search supports URL-safe base64 encoding and normal base64 encoding. A string that is base64 encoded during indexing should be decoded later with the same encoding options, or else the result won't match the original.

If the `useHttpServerUtilityUrlTokenEncode` or `useHttpServerUtilityUrlTokenDecode` parameters for encoding and decoding respectively are set to `true`, then `base64Encode` behaves like HttpServerUtility.UrlTokenEncode and `base64Decode` behaves like HttpServerUtility.UrlTokenDecode.

> ⚠ **Warning**
>
> If `base64Encode` is used to produce key values, `useHttpServerUtilityUrlTokenEncode` must be set to true. Only URL-safe base64 encoding can be used for key values. See **Naming rules** for the full set of restrictions on characters in key values.

The .NET libraries in Azure Cognitive Search assume the full .NET Framework, which provides built-in encoding. The `useHttpServerUtilityUrlTokenEncode` and `useHttpServerUtilityUrlTokenDecode` options apply this built-in functionality. If you're using .NET Core or another framework, we recommend setting those options to `false` and calling your framework's encoding and decoding functions directly.

The following table compares different base64 encodings of the string `00>00?00`. To determine the required processing (if any) for your base64 functions, apply your library encode function on the string `00>00?00` and compare the output with the expected output `MDA-MDA_MDA`.

Encoding	Base64 encode output	Extra processing after library encoding	Extra processing before library decoding
Base64 with padding	`MDA+MDA/MDA=`	Use URL-safe characters and remove padding	Use standard base64 characters and add padding
Base64 without padding	`MDA+MDA/MDA`	Use URL-safe characters	Use standard base64 characters
URL-safe base64 with padding	`MDA-MDA_MDA=`	Remove padding	Add padding
URL-safe base64 without padding	`MDA-MDA_MDA`	None	None

extractTokenAtPosition function

Splits a string field using the specified delimiter, and picks the token at the specified position in the resulting split.

This function uses the following parameters:

- `delimiter`: a string to use as the separator when splitting the input string.
- `position`: an integer zero-based position of the token to pick after the input string is split.

For example, if the input is `Jane Doe`, the `delimiter` is `" "`(space) and the `position` is 0, the result is `Jane`; if the `position` is 1, the result is `Doe`. If the position refers to a token that doesn't exist, an error is returned.

Example - extract a name

Your data source contains a `PersonName` field, and you want to index it as two separate `FirstName` and `LastName` fields. You can use this function to split the input using the space character as the delimiter.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "PersonName",
    "targetFieldName" : "FirstName",
    "mappingFunction" : { "name" : "extractTokenAtPosition", "parameters" :
{ "delimiter" : " ", "position" : 0 } }
  },
```

```
  {
    "sourceFieldName" : "PersonName",
    "targetFieldName" : "LastName",
    "mappingFunction" : { "name" : "extractTokenAtPosition", "parameters" :
{ "delimiter" : " ", "position" : 1 } }
  }]
```

jsonArrayToStringCollection function

Transforms a string formatted as a JSON array of strings into a string array that can be used to populate a `Collection(Edm.String)` field in the index.

For example, if the input string is `["red", "white", "blue"]`, then the target field of type `Collection(Edm.String)` will be populated with the three values `red`, `white`, and `blue`. For input values that can't be parsed as JSON string arrays, an error is returned.

Example - populate collection from relational data

Azure SQL Database doesn't have a built-in data type that naturally maps to `Collection(Edm.String)` fields in Azure Cognitive Search. To populate string collection fields, you can pre-process your source data as a JSON string array and then use the `jsonArrayToStringCollection` mapping function.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "tags",
    "mappingFunction" : { "name" : "jsonArrayToStringCollection" }
  }]
```

urlEncode function

This function can be used to encode a string so that it is "URL safe". When used with a string that contains characters that aren't allowed in a URL, this function will convert those "unsafe" characters into character-entity equivalents. This function uses the UTF-8 encoding format.

Example - document key lookup

`urlEncode` function can be used as an alternative to the `base64Encode` function, if only URL unsafe characters are to be converted, while keeping other characters as-is.

Say, the input string is `<hello>` - then the target field of type `(Edm.String)` will be populated with the value `%3chello%3e`

When you retrieve the encoded key at search time, you can then use the `urlDecode` function to get the original key value, and use that to retrieve the source document.

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "SourceKey",
    "targetFieldName" : "IndexKey",
    "mappingFunction" : {
      "name" : "urlEncode"
    }
  }]
```

urlDecode function

This function converts a URL-encoded string into a decoded string using UTF-8 encoding format.

Example - decode blob metadata

Some Azure storage clients automatically URL-encode blob metadata if it contains non-ASCII characters. However, if you want to make such metadata searchable (as plain text), you can use the `urlDecode` function to turn the encoded data back into regular strings when populating your search index.

JSON

```
"fieldMappings" : [
 {
   "sourceFieldName" : "UrlEncodedMetadata",
   "targetFieldName" : "SearchableMetadata",
   "mappingFunction" : {
     "name" : "urlDecode"
   }
 }]
```

fixedLengthEncode function

This function converts a string of any length to a fixed-length string.

Example - map document keys that are too long

When errors occur that are related to document key length exceeding 1024 characters, this function can be applied to reduce the length of the document key.

JSON

```
"fieldMappings" : [
 {
   "sourceFieldName" : "metadata_storage_path",
   "targetFieldName" : "your key field",
   "mappingFunction" : {
     "name" : "fixedLengthEncode"
   }
 }]
```

See also

- Supported data types in Cognitive Search
- SQL data type map

Content metadata properties used in Azure Cognitive Search

Article • 02/09/2023 • 2 minutes to read

Several of the indexer-supported data sources, including Azure Blob Storage, Azure Data Lake Storage Gen2, and SharePoint, contain standalone files or embedded objects of various content types. Many of those content types have metadata properties that can be useful to index. Just as you can create search fields for standard blob properties like `metadata_storage_name`, you can create fields in a search index for metadata properties that are specific to a document format.

Supported document formats

Cognitive Search supports blob indexing and SharePoint document indexing for the following document formats:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

Properties by document format

The following table summarizes processing done for each document format, and describes the metadata properties extracted by a blob indexer and the SharePoint indexer.

Document format / content type	Extracted metadata	Processing details

Document format / content type	Extracted metadata	Processing details
CSV (text/csv)	`metadata_content_type` `metadata_content_encoding`	Extract text NOTE: If you need to extract multiple document fields from a CSV blob, see Indexing CSV blobs for details

DOC (application/msword)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Extract text, including embedded documents

DOCM (application/vnd.ms-word.document.macroenabled.12)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Extract text, including embedded documents

DOCX (application/vnd.openxmlformats-officedocument.wordprocessingml.document)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Extract text, including embedded documents

EML (message/rfc822)	`metadata_content_type`
	`metadata_message_from`
	`metadata_message_to`
	`metadata_message_cc`
	`metadata_creation_date`
	`metadata_subject`
		Extract text, including attachments

EPUB (application/epub+zip)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_title`
	`metadata_description`
	`metadata_language`
	`metadata_keywords`
	`metadata_identifier`
	`metadata_publisher`
		Extract text from all documents in the archive

GZ (application/gzip)	`metadata_content_type`	Extract text from all documents in the archive

Document format / content type	Extracted metadata	Processing details

HTML (text/html or application/xhtml+xml)	`metadata_content_encoding`
	`metadata_content_type`
	`metadata_language`
	`metadata_description`
	`metadata_keywords`
	`metadata_title`
		Strip HTML markup and extract text

JSON (application/json)	`metadata_content_type` `metadata_content_encoding`	Extract text NOTE: If you need to extract multiple document fields from a JSON blob, see Indexing JSON blobs for details

KML (application/vnd.google-earth.kml+xml)	`metadata_content_type`
	`metadata_content_encoding`
	`metadata_language`
		Strip XML markup and extract text

MSG (application/vnd.ms-outlook)	`metadata_content_type`
	`metadata_message_from`
	`metadata_message_from_email`
	`metadata_message_to`
	`metadata_message_to_email`
	`metadata_message_cc`
	`metadata_message_cc_email`
	`metadata_message_bcc`
	`metadata_message_bcc_email`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_subject`
		Extract text, including text extracted from attachments. `metadata_message_to_email`, `metadata_message_cc_email` and `metadata_message_bcc_email` are string collections, the rest of the fields are strings.

ODP (application/vnd.oasis.opendocument.presentation)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_title`
		Extract text, including embedded documents

ODS (application/vnd.oasis.opendocument.spreadsheet)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
		Extract text, including embedded documents

ODT (application/vnd.oasis.opendocument.text)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Extract text, including embedded documents

Document format / content type	Extracted metadata	Processing details

PDF (application/pdf)	`metadata_content_type`
	`metadata_language`
	`metadata_author`
	`metadata_title`
	`metadata_creation_date`
		Extract text, including embedded documents (excluding images)

Plain text (text/plain)	`metadata_content_type`
	`metadata_content_encoding`
	`metadata_language`
		Extract text

PPT (application/vnd.ms-powerpoint)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_slide_count`
	`metadata_title`
		Extract text, including embedded documents

PPTM (application/vnd.ms-powerpoint.presentation.macroenabled.12)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_slide_count`
	`metadata_title`
		Extract text, including embedded documents

PPTX (application/vnd.openxmlformats-officedocument.presentationml.presentation)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_slide_count`
	`metadata_title`
		Extract text, including embedded documents

RTF (application/rtf)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Extract text

WORD 2003 XML (application/vnd.ms-wordml)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
		Strip XML markup and extract text

WORD XML (application/vnd.ms-word2006ml)	`metadata_content_type`
	`metadata_author`
	`metadata_character_count`
	`metadata_creation_date`
	`metadata_last_modified`
	`metadata_page_count`
	`metadata_word_count`
		Strip XML markup and extract text

Document format / content type	Extracted metadata	Processing details

XLS (application/vnd.ms-excel)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
		Extract text, including embedded documents

XLSM (application/vnd.ms-excel.sheet.macroenabled.12)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
		Extract text, including embedded documents

XLSX (application/vnd.openxmlformats-officedocument.spreadsheetml.sheet)	`metadata_content_type`
	`metadata_author`
	`metadata_creation_date`
	`metadata_last_modified`
		Extract text, including embedded documents

XML (application/xml)	`metadata_content_type`
	`metadata_content_encoding`
	`metadata_language`
		Strip XML markup and extract text

ZIP (application/zip)	`metadata_content_type`	Extract text from all documents in the archive

See also

- Indexers in Azure Cognitive Search
- AI enrichment overview
- Blob indexing overview
- SharePoint indexing

Additional resources

Documentation

Index blobs containing multiple documents - Azure Cognitive Search

Crawl Azure blobs for text content using the Azure Cognitive Search Blob indexer, where each blob might yield one or more search index documents.

Index operations using Azure Cognitive Search REST APIs

Learn which REST APIs are used to create, delete, and update Azure Cognitive Search index definitions.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Create Indexer (Azure Cognitive Search REST API-version=2020-06-30)

Indexers are resources that automate many aspects of data ingestion into an Azure Cognitive Search indexes. You must use a supported Azure data source to use this API.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Get Index (Azure Cognitive Search REST API)

Return an index definition in JSON, but not its contents, from an Azure Cognitive Search service.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Show 5 more

Indexing blobs and files to produce multiple search documents

Article • 01/31/2023 • 4 minutes to read

Applies to: Blob indexers, File indexers

By default, an indexer will treat the contents of a blob or file as a single search document. If you want a more granular representation in a search index, you can set **parsingMode** values to create multiple search documents from one blob or file. The **parsingMode** values that result in many search documents include `delimitedText` (for CSV), and `jsonArray` or `jsonLines` (for JSON).

When you use any of these parsing modes, the new search documents that emerge must have unique document keys, and a problem arises in determining where that value comes from. The parent blob has at least one unique value in the form of `metadata_storage_path property`, but if it contributes that value to more than one search document, the key is no longer unique in the index.

To address this problem, the blob indexer generates an `AzureSearch_DocumentKey` that uniquely identifies each child search document created from the single blob parent. This article explains how this feature works.

One-to-many document key

Each document that shows up in an Azure Cognitive Search index is uniquely identified by a document key.

When no parsing mode is specified, and if there's no explicit field mapping in the indexer definition for the search document key, the blob indexer automatically maps the `metadata_storage_path property` as the document key. This mapping ensures that each blob appears as a distinct search document, and it saves you the step of having to create this field mapping yourself (normally, only fields having identical names and types are automatically mapped).

When using any of the parsing modes, one blob maps to "many" search documents, making a document key solely based on blob metadata unsuitable. To overcome this constraint, Azure Cognitive Search is capable of generating a "one-to-many" document key for each individual entity extracted from a blob. This property is named AzureSearch_DocumentKey and is added to each individual entity extracted from the

blob. The value of this property is guaranteed to be unique for each individual entity across blobs and the entities will show up as separate search documents.

By default, when no explicit field mappings for the key index field are specified, the `AzureSearch_DocumentKey` is mapped to it, using the `base64Encode` field-mapping function.

Example

Assume you've an index definition with the following fields:

- `id`
- `temperature`
- `pressure`
- `timestamp`

And your blob container has blobs with the following structure:

Blob1.json

JSON

```
{ "temperature": 100, "pressure": 100, "timestamp": "2020-02-13T00:00:00Z" }
{ "temperature" : 33, "pressure" : 30, "timestamp": "2020-02-14T00:00:00Z" }
```

Blob2.json

JSON

```
{ "temperature": 1, "pressure": 1, "timestamp": "2019-01-12T00:00:00Z" }
{ "temperature" : 120, "pressure" : 3, "timestamp": "2017-05-11T00:00:00Z" }
```

When you create an indexer and set the **parsingMode** to `jsonLines` - without specifying any explicit field mappings for the key field, the following mapping will be applied implicitly.

HTTP

```
{
    "sourceFieldName" : "AzureSearch_DocumentKey",
    "targetFieldName": "id",
    "mappingFunction": { "name" : "base64Encode" }
}
```

This setup will result in disambiguated document keys, similar to the following illustration (base64-encoded ID shortened for brevity).

ID	temperature	pressure	timestamp
aHR0 ... YjEuanNvbjsx	100	100	2020-02-13T00:00:00Z
aHR0 ... YjEuanNvbjsy	33	30	2020-02-14T00:00:00Z
aHR0 ... YjIuanNvbjsx	1	1	2019-01-12T00:00:00Z
aHR0 ... YjIuanNvbjsy	120	3	2017-05-11T00:00:00Z

Custom field mapping for index key field

Assuming the same index definition as the previous example, suppose your blob container has blobs with the following structure:

Blob1.json

JSON

```
recordid, temperature, pressure, timestamp
1, 100, 100,"2019-02-13T00:00:00Z"
2, 33, 30,"2019-02-14T00:00:00Z"
```

Blob2.json

JSON

```
recordid, temperature, pressure, timestamp
1, 1, 1,"2018-01-12T00:00:00Z"
2, 120, 3,"2013-05-11T00:00:00Z"
```

When you create an indexer with `delimitedText` **parsingMode**, it might feel natural to set up a field-mapping function to the key field as follows:

HTTP

```
{
    "sourceFieldName" : "recordid",
    "targetFieldName": "id"
}
```

However, this mapping won't result in four documents showing up in the index because the `recordid` field isn't unique *across blobs*. Hence, we recommend you to make use of the implicit field mapping applied from the `AzureSearch_DocumentKey` property to the key index field for "one-to-many" parsing modes.

If you do want to set up an explicit field mapping, make sure that the *sourceField* is distinct for each individual entity **across all blobs**.

> ⓘ **Note**
>
> The approach used by `AzureSearch_DocumentKey` of ensuring uniqueness per extracted entity is subject to change and therefore you should not rely on it's value for your application's needs.

Specify the index key field in your data

Assuming the same index definition as the previous example and **parsingMode** is set to `jsonLines` without specifying any explicit field mappings so the mappings look like in the first example, suppose your blob container has blobs with the following structure:

Blob1.json

JSON

```
id, temperature, pressure, timestamp
1, 100, 100,"2019-02-13T00:00:00Z"
2, 33, 30,"2019-02-14T00:00:00Z"
```

Blob2.json

JSON

```
id, temperature, pressure, timestamp
1, 1, 1,"2018-01-12T00:00:00Z"
2, 120, 3,"2013-05-11T00:00:00Z"
```

Notice that each document contains the `id` field, which is defined as the `key` field in the index. In such a case, even though a document-unique `AzureSearch_DocumentKey` will be generated, it won't be used as the "key" for the document. Rather, the value of the `id` field will be mapped to the `key` field

Similar to the previous example, this mapping won't result in four documents showing up in the index because the `id` field isn't unique *across blobs*. When this is the case, any json entry that specifies an `id` will result in a merge on the existing document instead of an upload of a new document, and the state of the index will reflect the latest read entry with the specified `id`.

Next steps

If you aren't already familiar with the basic structure and workflow of blob indexing, you should review Indexing Azure Blob Storage with Azure Cognitive Search first. For more information about parsing modes for different blob content types, review the following articles.

Indexing CSV blobs | Indexing JSON blobs

Additional resources

Documentation

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Content metadata properties - Azure Cognitive Search

Metadata properties can provide content to fields in a search index. This article lists metadata properties supported in Azure Cognitive Search.

Search over CSV blobs - Azure Cognitive Search

Extract CSV blobs from Azure Blob Storage and import as search documents into Azure Cognitive Search using the delimitedText parsing mode.

Create Indexer (Azure Cognitive Search REST API-version=2020-06-30)

Indexers are resources that automate many aspects of data ingestion into an Azure Cognitive Search indexes. You must use a supported Azure data source to use this API.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Show 5 more

How to index plain text blobs and files in Azure Cognitive Search

Article • 09/15/2022 • 2 minutes to read

Applies to: Blob indexers, File indexers

When using an indexer to extract searchable blob text or file content for full text search, you can assign a parsing mode to get better indexing outcomes. By default, the indexer parses the content as a single chunk of text. However, if all blobs and files contain plain text in the same encoding, you can significantly improve indexing performance by using the `text` parsing mode.

Recommendations for use `text` parsing include:

- File type is .txt
- Files are of any type, but the content itself is text (for example, program source code, HTML, XML, and so forth). For files in a mark up language, any syntax characters will come through as static text.

Recall that all indexers serialize to JSON. By default, the contents of the entire text file will be indexed within one large field as `"content": "<file-contents>"`. Any new line and return instructions are embedded in the content field and expressed as `\r\n\`.

If you want a more granular outcome, and if the file type is compatible, consider the following solutions:

- delimitedText parsing mode, if the source is CSV
- jsonArray or jsonLines, if the source is JSON

A third option for breaking content into multiple parts requires advanced features in the form of AI enrichment. It adds analysis that identifies and assigns chunks of the file to different search fields. You might find a full or partial solution through built-in skills, but a more likely solution would be learning model that understands your content, articulated in custom learning model, wrapped in a custom skill.

Set up plain text indexing

To index plain text blobs, create or update an indexer definition with the `parsingMode` configuration property to `text` on a Create Indexer request:

```
HTTP
```

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  ... other parts of indexer definition
  "parameters" : { "configuration" : { "parsingMode" : "text" } }
}
```

By default, the `UTF-8` encoding is assumed. To specify a different encoding, use the `encoding` configuration property:

HTTP

```
{
  ... other parts of indexer definition
  "parameters" : { "configuration" : { "parsingMode" : "text", "encoding" : "windows-1252" } }
}
```

Request example

Parsing modes are specified in the indexer definition.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  "name" : "my-plaintext-indexer",
  "dataSourceName" : "my-blob-datasource",
  "targetIndexName" : "my-target-index",
  "parameters" : { "configuration" : { "parsingMode" : "delimitedText", "delimitedTextHeaders" : "id,datePublished,tags" } }
}
```

Next steps

- Indexers in Azure Cognitive Search
- How to configure a blob indexer
- Blob indexing overview

Index CSV blobs and files using delimitedText parsing mode

Article • 10/04/2022 • 2 minutes to read

Applies to: Blob indexers, File indexers

In Azure Cognitive Search, both blob indexers and file indexers support a `delimitedText` parsing mode for CSV files that treats each line in the CSV as a separate search document. For example, given the following comma-delimited text, the `delimitedText` parsing mode would result in two documents in the search index:

text

```
id, datePublished, tags
1, 2016-01-12, "azure-search,azure,cloud"
2, 2016-07-07, "cloud,mobile"
```

Without the `delimitedText` parsing mode, the entire contents of the CSV file would be treated as one search document.

Whenever you're creating multiple search documents from a single blob, be sure to review Indexing blobs to produce multiple search documents to understand how document key assignments work. The blob indexer is capable of finding or generating values that uniquely define each new document. Specifically, it can create a transitory `AzureSearch_DocumentKey` that generated when a blob is parsed into smaller parts, where the value is then used as the search document's key in the index.

Setting up CSV indexing

To index CSV blobs, create or update an indexer definition with the `delimitedText` parsing mode on a Create Indexer request:

HTTP

```
{
  "name" : "my-csv-indexer",
  ... other indexer properties
  "parameters" : { "configuration" : { "parsingMode" : "delimitedText",
"firstLineContainsHeaders" : true } }
}
```

`firstLineContainsHeaders` indicates that the first (non-blank) line of each blob contains headers. If blobs don't contain an initial header line, the headers should be specified in the indexer configuration:

```HTTP
"parameters" : { "configuration" : { "parsingMode" : "delimitedText",
"delimitedTextHeaders" : "id,datePublished,tags" } }
```

You can customize the delimiter character using the `delimitedTextDelimiter` configuration setting. For example:

```HTTP
"parameters" : { "configuration" : { "parsingMode" : "delimitedText",
"delimitedTextDelimiter" : "|" } }
```

> **Note**
>
> Currently, only the UTF-8 encoding is supported. If you need support for other encodings, vote for it on **UserVoice**.

> **Important**
>
> When you use the delimited text parsing mode, Azure Cognitive Search assumes that all blobs in your data source will be CSV. If you need to support a mix of CSV and non-CSV blobs in the same data source, please vote for it on **UserVoice**. Otherwise, considering using **file extension filters** to control which files are imported on each indexer run.

Request examples

Putting it all together, here are the complete payload examples.

Datasource:

```HTTP
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]
```

```
{
    "name" : "my-blob-datasource",
    "type" : "azureblob",
    "credentials" : { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=
<account key>;" },
    "container" : { "name" : "my-container", "query" : "<optional, my-
folder>" }
}
```

Indexer:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-
30
Content-Type: application/json
api-key: [admin key]

{
  "name" : "my-csv-indexer",
  "dataSourceName" : "my-blob-datasource",
  "targetIndexName" : "my-target-index",
  "parameters" : { "configuration" : { "parsingMode" : "delimitedText",
"delimitedTextHeaders" : "id,datePublished,tags" } }
}
```

See also

- Index data from Blob Storage
- Index data from File Storage

Additional resources

🕮 Documentation

Index blobs containing multiple documents - Azure Cognitive Search

Crawl Azure blobs for text content using the Azure Cognitive Search Blob indexer, where each blob might yield one or more search index documents.

Content metadata properties - Azure Cognitive Search

Metadata properties can provide content to fields in a search index. This article lists metadata properties supported in Azure Cognitive Search.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Show 5 more

Index JSON blobs and files in Azure Cognitive Search

Article • 03/23/2023 • 6 minutes to read

Applies to: Blob indexers, File indexers

This article shows you how to set JSON-specific properties for blobs or files that consist of JSON documents. JSON blobs in Azure Blob Storage or Azure File Storage commonly assume any of these forms:

- A single JSON document
- A JSON document containing an array of well-formed JSON elements
- A JSON document containing multiple entities, separated by a newline

The blob indexer provides a "parsingMode" parameter to optimize the output of the search document based on the structure. Parsing modes consist of the following options:

parsingMode	JSON document	Description
`json`	One per blob	(default) Parses JSON blobs as a single chunk of text. Each JSON blob becomes a single search document.
`jsonArray`	Multiple per blob	Parses a JSON array in the blob, where each element of the array becomes a separate search document.
`jsonLines`	Multiple per blob	Parses a blob that contains multiple JSON entities (also an array), with individual elements separated by a newline. The indexer starts a new search document after each new line.

For both `jsonArray` and `jsonLines`, you should review Indexing one blob to produce many search documents to understand how the blob indexer handles disambiguation of the document key for multiple search documents produced from the same blob.

Within the indexer definition, you can optionally set field mappings to choose which properties of the source JSON document are used to populate your target search index. For example, when using the `jsonArray` parsing mode, if the array exists as a lower-level property, you can set a "documentRoot" property indicating where the array is placed within the blob.

The following sections describe each mode in more detail. If you're unfamiliar with indexer clients and concepts, see Create a search indexer. You should also be familiar

with the details of basic blob indexer configuration, which isn't repeated here.

Index single JSON documents (one per blob)

By default, blob indexers parse JSON blobs as a single chunk of text, one search document for each blob in a container. If the JSON is structured, the search document can reflect that structure, with individual elements represented as individual fields. For example, assume you have the following JSON document in Azure Blob Storage:

HTTP

```
{
    "article" : {
        "text" : "A hopefully useful article explaining how to parse JSON
blobs",
        "datePublished" : "2020-04-13",
        "tags" : [ "search", "storage", "howto" ]
    }
}
```

The blob indexer parses the JSON document into a single search document, loading an index by matching "text", "datePublished", and "tags" from the source against identically named and typed target index fields. Given an index with "text", "datePublished, and "tags" fields, the blob indexer can infer the correct mapping without a field mapping present in the request.

Although the default behavior is one search document per JSON blob, setting the 'json' parsing mode changes the internal field mappings for content, promoting fields inside `content` to actual fields in the search index. An example indexer definition for the **`json`** parsing mode might look like this:

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-
30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "parameters" : { "configuration" : { "parsingMode" : "json" } }
}
```

> **Note**
>
> As with all indexers, if fields do not clearly match, you should expect to explicitly specify individual **field mappings** unless you are using the implicit fields mappings available for blob content and metadata, as described in **basic blob indexer configuration**.

json example (single hotel JSON files)

The hotel JSON document data set on GitHub is helpful for testing JSON parsing, where each blob represents a structured JSON file. You can upload the data files to Blob Storage and use the **Import data** wizard to quickly evaluate how this content is parsed into individual search documents.

The data set consists of five blobs, each containing a hotel document with an address collection and a rooms collection. The blob indexer detects both collections and reflects the structure of the input documents in the index schema.

Parse JSON arrays

Alternatively, you can use the JSON array option. This option is useful when blobs contain an array of well-formed JSON objects, and you want each element to become a separate search document. Using `jsonArrays`, the following JSON blob produces three separate documents, each with `"id"` and `"text"` fields.

text

```text
[
    { "id" : "1", "text" : "example 1" },
    { "id" : "2", "text" : "example 2" },
    { "id" : "3", "text" : "example 3" }
]
```

The "parameters" property on the indexer contains parsing mode values. For a JSON array, the indexer definition should look similar to the following example.

HTTP

```http
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]
```

```
{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "parameters" : { "configuration" : { "parsingMode" : "jsonArray" } }
}
```

jsonArrays example (clinical trials sample data)

The clinical trials JSON data set on GitHub is helpful for testing JSON array parsing. You can upload the data files to Blob storage and use the **Import data** wizard to quickly evaluate how this content is parsed into individual search documents.

The data set consists of eight blobs, each containing a JSON array of entities, for a total of 100 entities. The entities vary as to which fields are populated, but the end result is one search document per entity, from all arrays, in all blobs.

Parsing nested JSON arrays

For JSON arrays having nested elements, you can specify a "documentRoot" to indicate a multi-level structure. For example, if your blobs look like this:

HTTP

```
{
    "level1" : {
        "level2" : [
            { "id" : "1", "text" : "Use the documentRoot property" },
            { "id" : "2", "text" : "to pluck the array you want to index" },
            { "id" : "3", "text" : "even if it's nested inside the document"
}
        ]
    }
}
```

Use this configuration to index the array contained in the `level2` property:

HTTP

```
{
    "name" : "my-json-array-indexer",
    ... other indexer properties
    "parameters" : { "configuration" : { "parsingMode" : "jsonArray",
"documentRoot" : "/level1/level2" } }
}
```

Parse JSON entities separated by newlines

If your blob contains multiple JSON entities separated by a newline, and you want each element to become a separate search document, use `jsonLines`.

text

```text
{ "id" : "1", "text" : "example 1" }
{ "id" : "2", "text" : "example 2" }
{ "id" : "3", "text" : "example 3" }
```

For JSON lines, the indexer definition should look similar to the following example.

HTTP

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "parameters" : { "configuration" : { "parsingMode" : "jsonLines" } }
}
```

Map JSON fields to search fields

Field mappings are used to associate a source field with a destination field in situations where the field names and types are not identical. But field mappings can also be used to match parts of a JSON document and "lift" them into top-level fields of the search document.

The following example illustrates this scenario. For more information about field mappings in general, see field mappings.

HTTP

```HTTP
{
    "article" : {
        "text" : "A hopefully useful article explaining how to parse JSON blobs",
        "datePublished" : "2016-04-13"
        "tags" : [ "search", "storage", "howto" ]
```

```
    }
}
```

Assume a search index with the following fields: `text` of type `Edm.String`, `date` of type `Edm.DateTimeOffset`, and `tags` of type `Collection(Edm.String)`. Notice the discrepancy between "datePublished" in the source and `date` field in the index. To map your JSON into the desired shape, use the following field mappings:

```HTTP
"fieldMappings" : [
    { "sourceFieldName" : "/article/text", "targetFieldName" : "text" },
    { "sourceFieldName" : "/article/datePublished", "targetFieldName" :
"date" },
    { "sourceFieldName" : "/article/tags", "targetFieldName" : "tags" }
    ]
```

Source fields are specified using the JSON Pointer notation. You start with a forward slash to refer to the root of your JSON document, then pick the desired property (at arbitrary level of nesting) by using forward slash-separated path.

You can also refer to individual array elements by using a zero-based index. For example, to pick the first element of the "tags" array from the above example, use a field mapping like this:

```HTTP
{ "sourceFieldName" : "/article/tags/0", "targetFieldName" : "firstTag" }
```

> **Note**
>
> If "sourceFieldName" refers to a property that doesn't exist in the JSON blob, that mapping is skipped without an error. This behavior allows indexing to continue for JSON blobs that have a different schema (which is a common use case). Because there is no validation check, check the mappings carefully for typos so that you aren't losing documents for the wrong reason.

Next steps

- Configure blob indexers
- Define field mappings
- Indexers overview

- How to index CSV blobs with a blob indexer
- Tutorial: Search semi-structured data from Azure Blob Storage

Set up an indexer connection to Azure Storage using a managed identity

Article • 09/20/2022 • 3 minutes to read

This article explains how to set up an indexer connection to an Azure Storage account using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Azure Active Directory logins and require Azure role assignments to access data in Azure Storage.

> **Note**
>
> If storage is network-protected and in the same region as your search service, you must use a system-assigned managed identity and either one of the following network options: **connect as a trusted service**, or **connect using the resource instance rule**.

Prerequisites

- Create a managed identity for your search service.
- Assign a role in Azure Storage:
  - Choose **Storage Blob Data Reader** for data read access in Blob Storage and ADLS Gen2.
  - Choose **Reader and Data** for data read access in Table Storage and File Storage.
- You should be familiar with indexer concepts and configuration.

> **Tip**
>
> For a code example in C#, see **Index Data Lake Gen2 using Azure AD** on GitHub.

Create the data source

Create the data source and provide either a system-assigned managed identity or a user-assigned managed identity (preview).

System-assigned managed identity

The REST API, Azure portal, and the .NET SDK support using a system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You'll provide a ResourceId that has no account key or password. The ResourceId must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name.

Here's an example of how to create a data source to index data from a storage account using the Create Data Source REST API and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "blob-datasource",
    "type" : "azureblob",
    "credentials" : {
        "connectionString" : "ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource group name]/providers/Microsoft.Storage/storageAccounts/[storage account name]/;"
    },
    "container" : {
        "name" : "my-container", "query" : "<optional-virtual-directory-name>"
    }
}
```

User-assigned managed identity (preview)

The 2021-04-30-preview REST API supports connections based on a user-assigned managed identity. When you're connecting with a user-assigned managed identity, there are two changes to the data source definition:

- First, the format of the "credentials" property is a ResourceId that has no account key or password. The ResourceId must include the subscription ID of the storage

account, the resource group of the storage account, and the storage account name. This format is the same format as the system-assigned managed identity.

- Second, you'll add an "identity" property that contains the collection of user-assigned managed identities. Only one user-assigned managed identity should be provided when creating the data source. Set it to type "userAssignedIdentities".

Here's an example of how to create an indexer data source object using the preview Create or Update Data Source REST API:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2021-04-30-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "blob-datasource",
    "type" : "azureblob",
    "credentials" : {
        "connectionString" : "ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource group name]/providers/Microsoft.Storage/storageAccounts/[storage account name]/;"
    },
    "container" : {
        "name" : "my-container", "query" : "<optional-virtual-directory-name>"
    },
    "identity" : {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity" : "/subscriptions/[subscription ID]/resourcegroups/[resource group name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[managed identity name]"
    }
}
```

Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a Create Index REST API call with a searchable `content` field to store the text extracted from blobs:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
        "name" : "my-target-index",
        "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false }
        ]
}
```

Create the indexer

An indexer connects a data source with a target search index, and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create and run the indexer. If the indexer is successful, the connection syntax and role assignments are valid.

Here's a Create Indexer REST API call with a blob indexer definition. The indexer will run when you submit the request.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "blob-indexer",
    "dataSourceName" : "blob-datasource",
    "targetIndexName" : "my-target-index"
}
```

Accessing network secured data in storage accounts

Azure storage accounts can be further secured using firewalls and virtual networks. If you want to index content from a storage account that is secured using a firewall or virtual network, see Make indexer connections to Azure Storage as a trusted service.

See also

- Azure Blob indexer
- Azure Data Lake Storage Gen2 indexer
- Azure Table indexer
- C# Example: Index Data Lake Gen2 using Azure AD (GitHub)

Set up an indexer connection to Azure Cosmos DB via a managed identity

Article • 10/12/2022 • 4 minutes to read

This article explains how to set up an indexer connection to an Azure Cosmos DB database using a managed identity instead of providing credentials in the connection string.'

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Azure Active Directory logins and require Azure role assignments to access data in Azure Cosmos DB.

Prerequisites

- Create a managed identity for your search service.
- Assign a role in Azure Cosmos DB.

  For data reader access, you'll need the **Cosmos DB Account Reader** role and the identity used to make the request. This role works for all Azure Cosmos DB APIs supported by Cognitive Search. This is a control plane RBAC role.

  At this time, Cognitive Search obtains keys with the identity and uses those keys to connect to the Azure Cosmos DB account. This means that enforcing RBAC as the only authentication method in Azure Cosmos DB isn't supported when using Search with managed identities to connect to Azure Cosmos DB.
- You should be familiar with indexer concepts and configuration.

Create the data source

Create the data source and provide either a system-assigned managed identity or a user-assigned managed identity (preview) in the connection string.

System-assigned managed identity

The REST API, Azure portal, and the .NET SDK support using a system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You'll provide the database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure Cosmos DB, the resource group, and the Azure Cosmos DB account name.

- For SQL collections, the connection string doesn't require "ApiKind".
- For MongoDB collections, add "ApiKind=MongoDb" to the connection string and use a preview REST API.
- For Gremlin graphs, add "ApiKind=Gremlin" to the connection string and use a preview REST API.

Here's an example of how to create a data source to index data from a storage account using the Create Data Source REST API and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "[my-cosmosdb-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/[subscription-id]/resourceGroups/[rg-name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-name];Database=[cosmos-database];ApiKind=[SQL | Gremlin | MongoDB];"
    },
    "container": { "name": "[my-cosmos-collection]", "query": null },
    "dataChangeDetectionPolicy": null


}
```

User-assigned managed identity (preview)

The 2021-04-30-preview REST API supports connections based on a user-assigned managed identity. When you're connecting with a user-assigned managed identity, there are two changes to the data source definition:

- First, the format of the "credentials" property is the database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure Cosmos DB, the resource group, and the Azure Cosmos DB account name.
  - For SQL collections, the connection string doesn't require "ApiKind".
  - For MongoDB collections, add "ApiKind=MongoDb" to the connection string
  - For Gremlin graphs, add "ApiKind=Gremlin" to the connection string.

- Second, you'll add an "identity" property that contains the collection of user-assigned managed identities. Only one user-assigned managed identity should be provided when creating the data source. Set it to type "userAssignedIdentities".

Here's an example of how to create an indexer data source object using the preview Create or Update Data Source REST API:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2021-04-30-preview
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "[my-cosmosdb-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "ResourceId=/subscriptions/[subscription-id]/resourceGroups/[rg-name]/providers/Microsoft.DocumentDB/databaseAccounts/[cosmos-account-name];Database=[cosmos-database];ApiKind=[SQL | Gremlin | MongoDB];"
    },
    "container": {
        "name": "[my-cosmos-collection]", "query": null
    },
    "identity" : {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity": "/subscriptions/[subscription-id]/resourcegroups/[rg-name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[my-user-managed-identity-name]"
    },
    "dataChangeDetectionPolicy": null
}
```

Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a Create Index REST API call with a searchable `booktitle` field:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-target-index",
    "fields": [
    { "name": "id", "type": "Edm.String", "key": true, "searchable": false
},
    { "name": "booktitle", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false }
    ]
}
```

Create the indexer

An indexer connects a data source with a target search index and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create and run the indexer. If the indexer is successful, the connection syntax and role assignments are valid.

Here's a Create Indexer REST API call with an Azure Cosmos DB indexer definition. The indexer will run when you submit the request.

HTTP

```
    POST https://[service name].search.windows.net/indexers?api-
version=2020-06-30
    Content-Type: application/json
    api-key: [admin key]

    {
      "name" : "cosmos-db-indexer",
      "dataSourceName" : "cosmos-db-datasource",
      "targetIndexName" : "my-target-index"
    }
```

Troubleshooting

If you recently rotated your Azure Cosmos DB account keys, you'll need to wait up to 15 minutes for the managed identity connection string to work.

Check to see if the Azure Cosmos DB account has its access restricted to select networks. You can rule out any firewall issues by trying the connection without restrictions in place.

See also

- Indexing via Azure Cosmos DB for NoSQL
- Indexing via Azure Cosmos DB for MongoDB
- Indexing via Azure Cosmos DB for Apache Gremlin

Set up an indexer connection to Azure SQL using a managed identity

Article • 09/20/2022 • 5 minutes to read

This article explains how to set up an indexer connection to Azure SQL Database using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Azure Active Directory logins and require Azure role assignments to access data in Azure SQL.

Prerequisites

- Create a managed identity for your search service.
- Assign an Azure admin role on SQL. The identity used on the indexer connection needs read permissions. You must be an Azure AD admin with a server in SQL Database or SQL Managed Instance to grant read permissions on a database.
- You should be familiar with indexer concepts and configuration.

1 - Assign permissions to read the database

Follow the below steps to assign the search service or user-assigned managed identity permission to read the database.

1. Connect to Visual Studio

2. Authenticate with your Azure AD account

3. Execute the following commands:

Include the brackets around your search service name or user-assigned managed identity name.

```
CREATE USER [insert your search service name here or user-assigned
managed identity name] FROM EXTERNAL PROVIDER;
EXEC sp_addrolemember 'db_datareader', [insert your search service name
here or user-assigned managed identity name];
```

If you later change the search service identity or user-assigned identity after assigning permissions, you must remove the role membership and remove the user in the SQL

database, then repeat the permission assignment. Removing the role membership and user can be accomplished by running the following commands:

```
sp_droprolemember 'db_datareader', [insert your search service name or user-assigned managed identity name];

DROP USER IF EXISTS [insert your search service name or user-assigned managed identity name];
```

2 - Add a role assignment

In this section you'll, give your Azure Cognitive Search service permission to read data from your SQL Server. For detailed steps, see Assign Azure roles using the Azure portal.

1. In the Azure portal, navigate to your Azure SQL Server page.
2. Select **Access control (IAM)**.
3. Select **Add** > **Add role assignment**.

4. On the **Role** tab, select the appropriate **Reader** role.
5. On the **Members** tab, select **Managed identity**, and then select **Select members**.
6. Select your Azure subscription.
7. If you're using a system-assigned managed identity, select **System-assigned managed identity**, search for your search service, and then select it.
8. Otherwise, if you're using a user-assigned managed identity, select **User-assigned managed identity**, search for the name of the user-assigned managed identity,

and then select it.

9. On the **Review + assign** tab, select **Review + assign** to assign the role.

3 - Create the data source

Create the data source and provide either a system-assigned managed identity or a user-assigned managed identity (preview).

System-assigned managed identity

The REST API, Azure portal, and the .NET SDK support system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You'll provide an Initial Catalog or Database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure SQL Database, the resource group of SQL Database, and the name of the SQL database.

Here's an example of how to create a data source to index data from a storage account using the Create Data Source REST API and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

```
POST https://[service name].search.windows.net/datasources?api-version=2020-
06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database
name];ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource
group name]/providers/Microsoft.Sql/servers/[SQL Server name];Connection
Timeout=30;"
    },
    "container" : {
        "name" : "my-table"
    }
}
```

User-assigned managed identity (preview)

The 2021-04-30-preview REST API supports connections based on a user-assigned managed identity. When you're connecting with a user-assigned managed identity, there are two changes to the data source definition:

- First, the format of the "credentials" property is an Initial Catalog or Database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of Azure SQL Database, the resource group of SQL Database, and the name of the SQL database. This is the same format as the system-assigned managed identity.
- Second, you'll add an "identity" property that contains the collection of user-assigned managed identities. Only one user-assigned managed identity should be provided when creating the data source. Set it to type "userAssignedIdentities".

Here's an example of how to create an indexer data source object using the preview Create or Update Data Source REST API:

```
POST https://[service name].search.windows.net/datasources?api-version=2021-04-30-preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database name];ResourceId=/subscriptions/[subscription ID]/resourceGroups/[resource group name]/providers/Microsoft.Sql/servers/[SQL Server name];Connection Timeout=30;"
    },
    "container" : {
        "name" : "my-table"
    },
    "identity" : {
        "@odata.type": "#Microsoft.Azure.Search.DataUserAssignedIdentity",
        "userAssignedIdentity" : "/subscriptions/[subscription ID]/resourcegroups/[resource group name]/providers/Microsoft.ManagedIdentity/userAssignedIdentities/[managed identity name]"
    }
}
```

4 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a Create Index REST API call with a searchable `booktitle` field:

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-target-index",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "booktitle", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false }
    ]
}
```

5 - Create the indexer

An indexer connects a data source with a target search index, and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create the indexer. If the indexer is successful, the connection syntax and role assignments are valid.

Here's a Create Indexer REST API call with an Azure SQL indexer definition. The indexer will run when you submit the request.

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-indexer",
    "dataSourceName" : "sql-datasource",
    "targetIndexName" : "my-target-index"
```

Troubleshooting

If you get an error when the indexer tries to connect to the data source that says that the client isn't allowed to access the server, take a look at common indexer errors.

You can also rule out any firewall issues by trying the connection with and without restrictions in place.

See also

- Azure SQL indexer

Set up an indexer connection to Azure SQL Managed Instance using a managed identity

Article • 04/12/2023

This article describes how to set up an Azure Cognitive Search indexer connection to SQL Managed Instance using a managed identity instead of providing credentials in the connection string.

You can use a system-assigned managed identity or a user-assigned managed identity (preview). Managed identities are Azure AD logins and require Azure role assignments to access data in SQL Managed Instance.

Before learning more about this feature, it is recommended that you have an understanding of what an indexer is and how to set up an indexer for your data source. More information can be found at the following links:

- Indexer overview
- SQL Managed Instance indexer

Prerequisites

- Create a managed identity for your search service.
- Azure AD admin role on SQL Managed Instance:

  To assign read permissions on SQL Managed Instance, you must be an Azure Global Admin with a SQL Managed Instance. See Configure and manage Azure AD authentication with SQL Managed Instance and follow the steps to provision an Azure AD admin (SQL Managed Instance).
- Configure a public endpoint and network security group in SQL Managed Instance to allow connections from Azure Cognitive Search. If your Azure SQL Managed Instance is configured for private connections, create a shared private link in Cognitive Search to allow the connection.

1 - Assign permissions to read the database

Follow these steps to assign the search service system managed identity permission to read the SQL Managed database.

1. Connect to your SQL Managed Instance through SQL Server Management Studio (SSMS) by using one of the following methods:

    - Configure a point-to-site connection from on-premises
    - Configure an Azure VM

2. Authenticate with your Azure AD account.

3. From the left pane, locate the SQL database you will be using as data source for indexing and right-click it. Select **New Query**.

4. In the T-SQL window, copy the following commands and include the brackets around your search service name. Click on **Execute**.

Transact-SQL

```
CREATE USER [insert your search service name here or user-assigned
managed identity name] FROM EXTERNAL PROVIDER;
EXEC sp_addrolemember 'db_datareader', [insert your search service name
here or user-assigned managed identity name];
```

If you later change the search service system identity after assigning permissions, you must remove the role membership and remove the user in the SQL database, then repeat the permission assignment. Removing the role membership and user can be accomplished by running the following commands:

Transact-SQL

```
sp_droprolemember 'db_datareader', [insert your search service name or user-assigned managed identity name];

DROP USER IF EXISTS [insert your search service name or user-assigned managed identity name];
```

2 - Add a role assignment

In this step you will give your Azure Cognitive Search service permission to read data from your SQL Managed Instance.

1. In the Azure portal navigate to your SQL Managed Instance page.

2. Select **Access control (IAM)**.

3. Select **Add** then **Add role assignment**.

4. Select **Reader** role.

5. Leave **Assign access to** as **Azure AD user, group or service principal**.

6. If you're using a system-assigned managed identity, search for your search service, then select it. If you're using a user-assigned managed identity, search for the name of the user-assigned managed identity, then select it. Select **Save**.

Example for SQL Managed Instance using a system-assigned managed identity:

3 - Create the data source

Create the data source and provide a system-assigned managed identity.

System-assigned managed identity

The REST API, Azure portal, and the .NET SDK support system-assigned managed identity.

When you're connecting with a system-assigned managed identity, the only change to the data source definition is the format of the "credentials" property. You'll provide an

Initial Catalog or Database name and a ResourceId that has no account key or password. The ResourceId must include the subscription ID of SQL Managed Instance, the resource group of SQL Managed instance, and the name of the SQL database.

Here is an example of how to create a data source to index data from a storage account using the Create Data Source REST API and a managed identity connection string. The managed identity connection string format is the same for the REST API, .NET SDK, and the Azure portal.

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-mi-datasource",
    "type" : "azuresql",
    "credentials" : {
        "connectionString" : "Database=[SQL database name];ResourceId=/subscriptions/[subscription ID]/resourcegroups/[resource group name]/providers/Microsoft.Sql/managedInstances/[SQL Managed Instance name];Connection Timeout=100;"
    },
    "container" : {
        "name" : "my-table"
    }
}
```

4 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's a Create Index REST API call with a searchable `booktitle` field:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-target-index",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable":
```

```
false },
        { "name": "booktitle", "type": "Edm.String", "searchable": true,
"filterable": false, "sortable": false, "facetable": false }
    ]
}
```

5 - Create the indexer

An indexer connects a data source with a target search index, and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create the indexer.

Here's a Create Indexer REST API call with an Azure SQL indexer definition. The indexer will run when you submit the request.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-
30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sql-mi-indexer",
    "dataSourceName" : "sql-mi-datasource",
    "targetIndexName" : "my-target-index"
}
```

Troubleshooting

If you get an error when the indexer tries to connect to the data source that says that the client is not allowed to access the server, take a look at common indexer errors.

You can also rule out any firewall issues by trying the connection with and without restrictions in place.

See also

- SQL Managed Instance and Azure SQL Database indexer

Indexer troubleshooting guidance for Azure Cognitive Search

Article • 04/05/2023 • 13 minutes to read

Occasionally, indexers run into problems and there is no error to help with diagnosis. This article covers problems and potential resolutions when indexer results are unexpected and there is limited information to go on. If you have an error to investigate, see Troubleshooting common indexer errors and warnings instead.

Troubleshoot connections to restricted resources

For data sources that are secured by Azure network security mechanisms, indexers have a limited set of options for making the connection. Currently, indexers can access restricted data sources behind an IP firewall or on a virtual network through a private endpoint.

Firewall rules

Azure Storage, Azure Cosmos DB and Azure SQL provide a configurable firewall. There's no specific error message when the firewall is enabled. Typically, firewall errors are generic. Some common errors include:

- `The remote server returned an error: (403) Forbidden`
- `This request is not authorized to perform this operation`
- `Credentials provided in the connection string are invalid or have expired`

There are two options for allowing indexers to access these resources in such an instance:

- Disable the firewall, by allowing access from **All Networks** (if feasible).
- Alternatively, you can allow access for the IP address of your search service and the IP address range of `AzureCognitiveSearch` service tag in the firewall rules of your resource (IP address range restriction).

Details for configuring IP address range restrictions for each data source type can be found from the following links:

- Azure Storage

- Azure Cosmos DB
- Azure SQL

Limitation: IP address range restrictions only work if your search service, and your storage account are in different regions.

Azure functions (that could be used as a Custom Web Api skill) also support IP address restrictions. The list of IP addresses to configure would be the IP address of your search service and the IP address range of `AzureCognitiveSearch` service tag.

For more information about connecting to a virtual machine, see Configure a connection to SQL Server on an Azure VM.

Configure network security group (NSG) rules

When accessing data in a SQL managed instance, or when an Azure VM is used as the web service URI for a Custom Web Api skill, customers need not be concerned with specific IP addresses.

In such cases, the Azure VM, or the SQL managed instance can be configured to reside within a virtual network. Then a network security group can be configured to filter the type of network traffic that can flow in and out of the virtual network subnets and network interfaces.

The `AzureCognitiveSearch` service tag can be directly used in the inbound NSG rules without needing to look up its IP address range.

More details for accessing data in a SQL managed instance are outlined here.

Network errors

Usually, network errors are generic. Some common errors include:

- `A network-related or instance-specific error occurred while establishing a connection to the server`
- `The server was not found or was not accessible`
- `Verify that the instance name is correct and that the source is configured to allow remote connections`

When you are receiving any of those errors:

- Make sure your source is accessible by trying to connect to it directly and not through the search service

- Check your source in the Azure portal for any current errors or outages
- Check for any network outages in Azure Status
- Check you are using public DNS for name resolution and not an Azure Private DNS

Azure SQL Database serverless indexing (error code 40613)

If your SQL database is on a serverless compute tier, make sure that the database is running (and not paused) when the indexer connects to it.

If the database is paused, the first login from your search service is expected to auto-resume the database, but returning an error stating that the database is unavailable with error code 40613. After the database is running, retry the login to establish connectivity.

Azure Active Directory Conditional Access policies

When creating a SharePoint indexer, you will go through a step that requires you to sign in to your Azure AD app after providing a device code. If you receive a message that says `"Your sign-in was successful but your admin requires the device requesting access to be managed"` the indexer is likely being blocked from accessing the SharePoint document library due to a Conditional Access policy.

To update the policy to allow the indexer access to the document library, follow the below steps:

1. Open the Azure portal and search **Azure AD Conditional Access**, then select **Policies** on the left menu. If you don't have access to view this page, you need to either find someone who has access or get access.

2. Determine which policy is blocking the SharePoint indexer from accessing the document library. The policy that might be blocking the indexer includes the user account that you used to authenticate during the indexer creation step in the **Users and groups** section. The policy also might have **Conditions** that:

    - Restrict **Windows** platforms.
    - Restrict **Mobile apps and desktop clients**.
    - Have **Device state** configured to **Yes**.

3. Once you've confirmed there is a policy that is blocking the indexer, you next need to make an exemption for the indexer. Retrieve the search service IP address.

a. Obtain the fully qualified domain name (FQDN) of your search service. The FQDN looks like `<search-service-name>.search.windows.net`. You can find out the FQDN by looking up your search service on the Azure portal.

The IP address of the search service can be obtained by performing a `nslookup` (or a `ping`) of the FQDN. In the following example, you would add "150.0.0.1" to an inbound rule on the Azure Storage firewall. It might take up to 15 minutes after the firewall settings have been updated for the search service indexer to be able to access the Azure Storage account.

Azure PowerShell

```
nslookup contoso.search.windows.net
Server:  server.example.org
Address:  10.50.10.50

Non-authoritative answer:
Name:    <name>
Address:  150.0.0.1
Aliases:  contoso.search.windows.net
```

4. Get the IP address ranges for the indexer execution environment for your region.

   Extra IP addresses are used for requests that originate from the indexer's multi-tenant execution environment. You can get this IP address range from the service tag.

The IP address ranges for the `AzureCognitiveSearch` service tag can be either obtained via the discovery API or the downloadable JSON file.

For this walkthrough, assuming the search service is the Azure Public cloud, the Azure Public JSON file should be downloaded.

From the JSON file, assuming the search service is in West Central US, the list of IP addresses for the multi-tenant indexer execution environment are listed below.

JSON

```
{
  "name": "AzureCognitiveSearch.WestCentralUS",
  "id": "AzureCognitiveSearch.WestCentralUS",
  "properties": {
    "changeNumber": 1,
    "region": "westcentralus",
    "platform": "Azure",
    "systemService": "AzureCognitiveSearch",
    "addressPrefixes": [
      "52.150.139.0/26",
      "52.253.133.74/32"
    ]
  }
}
```

5. Back on the Conditional Access page in Azure portal, select **Named locations** from the menu on the left, then select **+ IP ranges location**. Give your new named location a name and add the IP ranges for your search service and indexer execution environments that you collected in the last two steps.

- For your search service IP address, you may need to add "/32" to the end of the IP address since it only accepts valid IP ranges.
- Remember that for the indexer execution environment IP ranges, you only need to add the IP ranges for the region that your search service is in.

6. Exclude the new Named location from the policy.
   a. Select **Policies** on the left menu.
   b. Select the policy that is blocking the indexer.
   c. Select **Conditions**.
   d. Select **Locations**.
   e. Select **Exclude** then add the new Named location.
   f. **Save** the changes.

7. Wait a few minutes for the policy to update and enforce the new policy rules.

8. Attempt to create the indexer again
   a. Send an update request for the data source object that you created.
   b. Resend the indexer create request. Use the new code to sign in, then send another indexer creation request.

Indexing unsupported document types

If you are indexing content from Azure Blob Storage, and the container includes blobs of an unsupported content type, the indexer skips that document. In other cases, there may be problems with individual documents.

You can set configuration options to allow indexer processing to continue in the event of problems with individual documents.

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  ... other parts of indexer definition
  "parameters" : { "configuration" : { "failOnUnsupportedContentType" : false, "failOnUnprocessableDocument" : false } }
}
```

Missing documents

Indexers extract documents or rows from an external data source and create *search documents*, which are then indexed by the search service. Occasionally, a document that exists in data source fails to appear in a search index. This unexpected result can occur due to the following reasons:

- The document was updated after the indexer was run. If your indexer is on a schedule, it eventually reruns and picks up the document.
- The indexer timed out before the document could be ingested. There are maximum processing time limits after which no documents are processed. You can check indexer status in the portal or by calling Get Indexer Status (REST API).
- Field mappings or AI enrichment have changed the document and its articulation in the search index is different from what you expect.
- Change tracking values are erroneous or prerequisites are missing. If your high watermark value is a date set to a future time, then any documents that have a date less than this are skipped by the indexer. You can understand your indexer's change tracking state using the 'initialTrackingState' and 'finalTrackingState' fields in the indexer status. Indexers for Azure SQL and MySQL must have an index on the high water mark column of the source table, or queries used by the indexer may time out.

> **Tip**
>
> If documents are missing, check the **query** you are using to make sure it isn't excluding the document in question. To query for a specific document, use the **Lookup Document REST API**.

Missing content from Blob Storage

The blob indexer finds and extracts text from blobs in a container. Some problems with extracting text include:

- The document only contains scanned images. PDF blobs that have non-text content, such as scanned images (JPGs), don't produce results in a standard blob indexing pipeline. If you have image content with text elements, you can use OCR or image analysis to find and extract the text.

- The blob indexer is configured to only index metadata. To extract content, the blob indexer must be configured to extract both content and metadata:

```
HTTP
```

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  ... other parts of indexer definition
  "parameters" : { "configuration" : { "dataToExtract" : "contentAndMetadata" } }
}
```

Missing content from Azure Cosmos DB

Azure Cognitive Search has an implicit dependency on Azure Cosmos DB indexing. If you turn off automatic indexing in Azure Cosmos DB, Azure Cognitive Search returns a successful state, but fails to index container contents. For instructions on how to check settings and turn on indexing, see Manage indexing in Azure Cosmos DB.

Indexer reflects a different document count than data source or index

Indexer may show a different document count than either the data source, the index or count in your code, depending on specific circumstances. Here are some possible causes of why this behavior may occur:

- The indexer has a Deleted Document Policy. The deleted documents get counted on the indexer end if they are indexed before they get deleted.
- If the ID column in the data source is not unique. This applies to data sources that have the concept of columns, such as Azure Cosmos DB.
- If the data source definition has a different query than the one you are using to estimate the number of records. In example, in your data base you are querying all your data base record count, while in the data source definition query you may be selecting just a subset of records to index.
- The counts are being checked in different intervals for each component of the pipeline: data source, indexer and index.
- The index may take some minutes to show the real document count.
- The data source has a file that's mapped to many documents. This condition can occur when indexing blobs and "parsingMode" is set to `jsonArray` and `jsonLines`.
- Due to documents processed multiple times.

Documents processed multiple times

Indexers use a conservative buffering strategy to ensure that every new and changed document in the data source is picked up during indexing. In certain situations, these buffers can overlap, causing an indexer to index a document two or more times resulting in the processed documents count to be more than actual number of documents in the data source. This behavior does **not** affect the data stored in the index, such as duplicating documents, only that it may take longer to reach eventual consistency. This condition can be especially prevalent if any of the following criteria are true:

- On-demand indexer requests are issued in quick succession
- The data source's topology includes multiple replicas and partitions (one such example is discussed here)
- The data source is an Azure SQL database and the column chosen as "high water mark" is of type `datetime2`

Indexers are not intended to be invoked multiple times in quick succession. If you need updates quickly, the supported approach is to push updates to the index while simultaneously updating the data source. For on-demand processing, we recommend that you pace your requests in five-minute intervals or more, and run the indexer on a schedule.

Example of duplicate document processing with 30 second buffer

Conditions under which a document is processed twice is explained in the following timeline that notes each action and counter action. The following timeline illustrates the issue:

Timeline (hh:mm:ss)	Event	Indexer High Water Mark	Comment
00:01:00	Write `doc1` to data source with eventual consistency	`null`	Document timestamp is 00:01:00.
00:01:05	Write `doc2` to data source with eventual consistency	`null`	Document timestamp is 00:01:05.
00:01:10	Indexer starts	`null`	

Timeline (hh:mm:ss)	Event	Indexer High Water Mark	Comment
00:01:11	Indexer queries for all changes before 00:01:10; the replica that the indexer queries happens to be only aware of `doc2`; only `doc2` is retrieved	`null`	Indexer requests all changes before starting timestamp but actually receives a subset. This behavior necessitates the look back buffer period.
00:01:12	Indexer processes `doc2` for the first time	`null`	
00:01:13	Indexer ends	00:01:10	High water mark is updated to starting timestamp of current indexer execution.
00:01:20	Indexer starts	00:01:10	
00:01:21	Indexer queries for all changes between 00:00:40 and 00:01:20; the replica that the indexer queries happens to be aware of both `doc1` and `doc2`; retrieves `doc1` and `doc2`	00:01:10	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:22	Indexer processes `doc1` for the first time	00:01:10	
00:01:23	Indexer processes `doc2` for the second time	00:01:10	
00:01:24	Indexer ends	00:01:20	High water mark is updated to starting timestamp of current indexer execution.
00:01:32	Indexer starts	00:01:20	
00:01:33	Indexer queries for all changes between 00:00:50 and 00:01:32; retrieves `doc1` and `doc2`	00:01:20	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:34	Indexer processes `doc1` for the second time	00:01:20	
00:01:35	Indexer processes `doc2` for the third time	00:01:20	

Timeline (hh:mm:ss)	Event	Indexer High Water Mark	Comment
00:01:36	Indexer ends	00:01:32	High water mark is updated to starting timestamp of current indexer execution.
00:01:40	Indexer starts	00:01:32	
00:01:41	Indexer queries for all changes between 00:01:02 and 00:01:40; retrieves `doc2`	00:01:32	Indexer requests for all changes between current high water mark minus the 30 second buffer, and starting timestamp of current indexer execution.
00:01:42	Indexer processes `doc2` for the fourth time	00:01:32	
00:01:43	Indexer ends	00:01:40	Notice this indexer execution started more than 30 seconds after the last write to the data source and also processed `doc2`. This is the expected behavior because if all indexer executions before 00:01:35 are eliminated, this becomes the first and only execution to process `doc1` and `doc2`.

In practice, this scenario only happens when on-demand indexers are manually invoked within minutes of each other, for certain data sources. It may result in mismatched numbers (like the indexer processed 345 documents total according to the indexer execution stats, but there are 340 documents in the data source and index) or potentially increased billing if you are running the same skills for the same document multiple times. Running an indexer using a schedule is the preferred recommendation.

Indexing documents with sensitivity labels

If you have sensitivity labels set on documents, you might not be able to index them. If you're getting errors, remove the labels prior to indexing.

See also

- Troubleshooting common indexer errors and warnings
- Monitor indexer-based indexing

Troubleshooting common indexer errors and warnings in Azure Cognitive Search

Article • 10/12/2022 • 23 minutes to read

This article provides information and solutions to common errors and warnings you might encounter during indexing and AI enrichment in Azure Cognitive Search.

Indexing stops when the error count exceeds 'maxFailedItems'.

If you want indexers to ignore these errors (and skip over "failed documents"), consider updating the `maxFailedItems` and `maxFailedItemsPerBatch` as described here.

> **Note**
>
> Each failed document along with its document key (when available) will show up as an error in the indexer execution status. You can utilize the **index api** to manually upload the documents at a later point if you have set the indexer to tolerate failures.

The error information in this article can help you resolve errors, allowing indexing to continue.

Warnings don't stop indexing, but they do indicate conditions that could result in unexpected outcomes. Whether you take action or not depends on the data and your scenario.

Beginning with API version `2019-05-06`, item-level Indexer errors and warnings are structured to provide increased clarity around causes and next steps. They contain the following properties:

Property	Description	Example
Key	The document ID of the document impacted by the error or warning.	`https://<storageaccount>.blob.core.windows.net/jfk-1k/docid-32112954.pdf`

Name	The operation name describing where the error or warning occurred. This is generated by the following structure: `[category].[subcategory].[resourceType].[resourceName]`	`DocumentExtraction.azureblob.myBlobContainerName`
		`Enrichment.WebApiSkill.mySkillName`
		`Projection.SearchIndex.OutputFieldMapping.myOutputFieldName`
		`Projection.SearchIndex.MergeOrUpload.myIndexName`
		`Projection.KnowledgeStore.Table.myTableName`

Message	A high-level description of the error or warning.	`Could not execute skill because the Web Api request failed.`
Details	Any additional details which may be helpful to diagnose the issue, such as the WebApi response if executing a custom skill failed.	`link-cryptonyms-list - Error processing the request record : System.ArgumentNullException: Value cannot be null. Parameter name: source at System.Linq.Enumerable.All[TSource](IEnumerable 1 source, Func 2 predicate) at Microsoft.CognitiveSearch.WebApiSkills.JfkWebApiSkills. ...rest of stack trace...`
DocumentationLink	A link to relevant documentation with detailed information to debug and resolve the issue. This link will often point to one of the below sections on this page.	`https://go.microsoft.com/fwlink/?linkid=2106475`

`Error: Could not read document`

Indexer was unable to read the document from the data source. This can happen due to:

Reason	Details/Example	Resolution

Reason	Details/Example	Resolution
Inconsistent field types across different documents	`Type of value has a mismatch with column type. Couldn't store '{47.6,-122.1}' in authors column. Expected type is JArray.` `Error converting data type nvarchar to float.` `Conversion failed when converting the nvarchar value '12 months' to data type int.` `Arithmetic overflow error converting expression to data type int.`	Ensure that the type of each field is the same across different documents. For example, if the first document `'startTime'` field is a DateTime, and in the second document it's a string, this error will be hit.
Errors from the data source's underlying service	From Azure Cosmos DB: `{"Errors":["Request rate is large"]}`	Check your storage instance to ensure it's healthy. You might need to adjust your scaling or partitioning.
Transient issues	`A transport-level error has occurred when receiving results from the server. (provider: TCP Provider, error: 0 - An existing connection was forcibly closed by the remote host`	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.

`Error: Could not extract content or metadata from your document`

Indexer with a Blob data source was unable to extract the content or metadata from the document (for example, a PDF file). This can happen due to:

Reason	Details/Example	Resolution
Blob is over the size limit	`Document is '150441598' bytes, which exceeds the maximum size '134217728' bytes for document extraction for your current service tier.`	Blob indexing errors
Blob has unsupported content type	`Document has unsupported content type 'image/png'`	Blob indexing errors
Blob is encrypted	`Document could not be processed - it may be encrypted or password protected.`	You can skip the blob with blob settings.
Transient issues	`Error processing blob: The request was aborted: The request was canceled.` `Document timed out during processing.`	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.

`Error: Could not parse document`

Indexer read the document from the data source, but there was an issue converting the document content into the specified field mapping schema. This can happen due to:

Reason	Details/Example	Resolution
The document key is missing	`Document key cannot be missing or empty`	Ensure all documents have valid document keys. The document key is determined by setting the 'key' property as part of the index definition. Indexers will emit this error when the property flagged as the 'key' cannot be found on a particular document.
The document key is invalid	`Document key cannot be longer than 1024 characters`	Modify the document key to meet the validation requirements.
Could not apply field mapping to a field	`Could not apply mapping function 'functionName' to field 'fieldName'. Array cannot be null. Parameter name: bytes`	Double check the field mappings defined on the indexer, and compare with the data of the specified field of the failed document. It may be necessary to modify the field mappings or the document data.

Reason	Details/Example	Resolution
Could not read field value	`Could not read the value of column 'fieldName' at index 'fieldIndex'. A transport-level error has occurred when receiving results from the server. (provider: TCP Provider, error: 0 - An existing connection was forcibly closed by the remote host.)`	These errors are typically due to unexpected connectivity issues with the data source's underlying service. Try running the document through your indexer again later.

`Error: Could not map output field 'xyz' to search index due to deserialization problem while applying mapping function 'abc'`

The output mapping might have failed because the output data is in the wrong format for the mapping function you're using. For example, applying Base64Encode mapping function on binary data would generate this error. To resolve the issue, either rerun indexer without specifying mapping function or ensure that the mapping function is compatible with the output field data type. See Output field mapping for details.

`Error: Could not execute skill`

The indexer was not able to run a skill in the skillset.

Reason	Details/Example	Resolution
Transient connectivity issues	A transient error occurred. Please try again later.	Occasionally there are unexpected connectivity issues. Try running the document through your indexer again later.
Potential product bug	An unexpected error occurred.	This indicates an unknown class of failure and may indicate a product bug. File a support ticket to get help.
A skill has encountered an error during execution	(From Merge Skill) One or more offset values were invalid and could not be parsed. Items were inserted at the end of the text	Use the information in the error message to fix the issue. This kind of failure will require action to resolve.

`Error: Could not execute skill because the Web API request failed`

The skill execution failed because the call to the Web API failed. Typically, this class of failure occurs when custom skills are used, in which case you'll need to debug your custom code to resolve the issue. If instead the failure is from a built-in skill, refer to the error message for help in fixing the issue.

While debugging this issue, be sure to pay attention to any skill input warnings for this skill. Your Web API endpoint may be failing because the indexer is passing it unexpected input.

`Error: Could not execute skill because Web API skill response is invalid`

The skill execution failed because the call to the Web API returned an invalid response. Typically, this class of failure occurs when custom skills are used, in which case you'll need to debug your custom code to resolve the issue. If instead the failure is from a built-in skill, file a support ticket to get assistance.

`Error: Type of value has a mismatch with column type. Couldn't store in 'xyz' column. Expected type is 'abc'`

If your data source has a field with a different data type than the field you're trying to map in your index, you may encounter this error. Check your data source field data types and make sure they are mapped correctly to your index data types.

`Error: Skill did not execute within the time limit`

There are two cases under which you may encounter this error message, each of which should be treated differently. Follow the instructions below depending on what skill returned this error for you.

Built-in Cognitive Service skills

Many of the built-in cognitive skills, such as language detection, entity recognition, or OCR, are backed by a Cognitive Service API endpoint. Sometimes there are transient issues with these endpoints and a request will time out. For transient issues, there is no remedy except to wait and try again. As a mitigation, consider setting your indexer to run on a schedule. Scheduled indexing picks up where it left off. Assuming transient issues are resolved, indexing and cognitive skill processing should be able to continue on the next scheduled run.

If you continue to see this error on the same document for a built-in cognitive skill, file a support ticket to get assistance, as this isn't expected.

Custom skills

If you encounter a timeout error with a custom skill, there are a couple of things you can try. First, review your custom skill and ensure that it's not getting stuck in an infinite loop and that it's returning a result consistently. Once you have confirmed that a result is returned, check the duration of execution. If you didn't explicitly set a `timeout` value on your custom skill definition, then the default `timeout` is 30 seconds. If 30 seconds isn't long enough for your skill to execute, you may specify a higher `timeout` value on your custom skill definition. Here's an example of a custom skill definition where the timeout is set to 90 seconds:

JSON

```
{
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
      "uri": "<your custom skill uri>",
      "batchSize": 1,
      "timeout": "PT90S",
      "context": "/document",
      "inputs": [
        {
          "name": "input",
          "source": "/document/content"
        }
      ],
      "outputs": [
        {
          "name": "output",
          "targetName": "output"
        }
      ]
    }
```

The maximum value that you can set for the `timeout` parameter is 230 seconds. If your custom skill is unable to execute consistently within 230 seconds, you may consider reducing the `batchSize` of your custom skill so that it will have fewer documents to process within a single execution. If you have already set your `batchSize` to 1, you'll need to rewrite the skill to be able to execute in under 230 seconds or otherwise split it into multiple custom skills so that the execution time for any single custom skill is a maximum of 230 seconds. Review the custom skill documentation for more information.

`Error: Could not 'MergeOrUpload' | 'Delete' document to the search index`

The document was read and processed, but the indexer could not add it to the search index. This can happen due to:

Reason	Details/Example	Resolution
A field contains a term that is too large	A term in your document is larger than the 32 KB limit	You can avoid this restriction by ensuring the field isn't configured as filterable, facetable, or sortable.
Document is too large to be indexed	A document is larger than the maximum api request size	How to index large data sets
Document contains too many objects in collection	A collection in your document exceeds the maximum elements across all complex collections limit. `The document with key '1000052' has '4303' objects in collections (JSON arrays). At most '3000' objects are allowed to be in collections across the entire document. Remove objects from collections and try indexing the document again.`	We recommend reducing the size of the complex collection in the document to below the limit and avoid high storage utilization.
Trouble connecting to the target index (that persists after retries) because the service is under other load, such as querying or indexing.	Failed to establish connection to update index. Search service is under heavy load.	Scale up your search service
Search service is being patched for service update, or is in the middle of a topology reconfiguration.	Failed to establish connection to update index. Search service is currently down/Search service is undergoing a transition.	Configure service with at least 3 replicas for 99.9% availability per SLA documentation
Failure in the underlying compute/networking resource (rare)	Failed to establish connection to update index. An unknown failure occurred.	Configure indexers to run on a schedule to pick up from a failed state.
An indexing request made to the target index was not acknowledged within a timeout period due to network issues.	Could not establish connection to the search index in a timely manner.	Configure indexers to run on a schedule to pick up from a failed state. Additionally, try lowering the indexer batch size if this error condition persists.

`Error: Could not index document because some of the document's data was not valid`

The document was read and processed by the indexer, but due to a mismatch in the configuration of the index fields and the data extracted and processed by the indexer, it could not be added to the search index. This can happen due to:

Reason	Details/Example
Data type of the field(s) extracted by the indexer is incompatible with the data model of the corresponding target index field.	`The data field '_data_' in the document with key '888' has an invalid value 'of type 'Edm.String''. The expected type was 'Collection(Edm.String)'.`
Failed to extract any JSON entity from a string value.	`Could not parse value 'of type 'Edm.String'' of field '_data_' as a JSON object.` `Error:'After parsing a value an unexpected character was encountered: ''. Path '_path_', line 1, position 3162.'`
Failed to extract a collection of JSON entities from a string value.	`Could not parse value 'of type 'Edm.String'' of field '_data_' as a JSON array.` `Error:'After parsing a value an unexpected character was encountered: ''. Path '[0]', line 1, position 27.'`

Reason	Details/Example
An unknown type was discovered in the source document.	`Unknown type '_unknown_' cannot be indexed`
An incompatible notation for geography points was used in the source document.	`WKT POINT string literals are not supported. Use GeoJson point literals instead`

In all these cases, refer to Supported Data types and Data type map for indexers to make sure that you build the index schema correctly and have set up appropriate indexer field mappings. The error message will include details that can help track down the source of the mismatch.

`Error: Integrated change tracking policy cannot be used because table has a composite primary key`

This applies to SQL tables, and usually happens when the key is either defined as a composite key or, when the table has defined a unique clustered index (as in a SQL index, not an Azure Search index). The main reason is that the key attribute is modified to be a composite primary key in the case of a unique clustered index. In that case, make sure that your SQL table doesn't have a unique clustered index, or that you map the key field to a field that is guaranteed not to have duplicate values.

`Error: Could not process document within indexer max run time`

This error occurs when the indexer is unable to finish processing a single document from the data source within the allowed execution time. Maximum running time is shorter when skillsets are used. When this error occurs, if you have maxFailedItems set to a value other than 0, the indexer bypasses the document on future runs so that indexing can progress. If you can't afford to skip any document, or if you're seeing this error consistently, consider breaking documents into smaller documents so that partial progress can be made within a single indexer execution.

`Error: Could not project document`

This error occurs when the indexer is attempting to project data into a knowledge store and there was a failure on the attempt. This failure could be consistent and fixable, or it could be a transient failure with the projection output sink that you may need to wait and retry in order to resolve. Here's a set of known failure states and possible resolutions.

Reason	Details/Example	Resolution
Could not update projection blob `'blobUri'` in container `'containerName'`	The specified container doesn't exist.	The indexer will check if the specified container has been previously created and will create it if necessary, but it only performs this check once per indexer run. This error means that something deleted the container after this step. To resolve this error, try this: leave your storage account information alone, wait for the indexer to finish, and then rerun the indexer.
Could not update projection blob `'blobUri'` in container `'containerName'`	Unable to write data to the transport connection: An existing connection was forcibly closed by the remote host.	This is expected to be a transient failure with Azure Storage and thus should be resolved by rerunning the indexer. If you encounter this error consistently, file a support ticket so it can be investigated further.
Could not update row `'projectionRow'` in table `'tableName'`	The server is busy.	This is expected to be a transient failure with Azure Storage and thus should be resolved by rerunning the indexer. If you encounter this error consistently, file a support ticket so it can be investigated further.

`Error: The cognitive service for skill '<skill-name>' has been throttled`

Skill execution failed because the call to Cognitive Services was throttled. Typically, this class of failure occurs when too many skills are executing in parallel. If you're using the Microsoft.Search.Documents client library to run the indexer, you can use the SearchIndexingBufferedSender to get automatic retry on failed steps. Otherwise, you can reset and rerun the indexer.

`Warning: Skill input was invalid`

An input to the skill was missing, it has the wrong type, or otherwise, invalid. The warning message will indicate the impact:

1. `Could not execute skill`

1. `Skill executed but may have unexpected results`

Cognitive skills have required inputs and optional inputs. For example, the Key phrase extraction skill has two required inputs `text`, `languageCode`, and no optional inputs. Custom skill inputs are all considered optional inputs.

If required inputs are missing or if the input isn't the right type, the skill gets skipped and generates a warning. Skipped skills don't generate outputs. If downstream skills consume the outputs of the skipped skill, they may generate additional warnings.

If an optional input is missing, the skill still runs but may produce unexpected output due to the missing input.

In both cases, this warning may be expected due to the shape of your data. For example, if you have a document containing information about people with the fields `firstName`, `middleName`, and `lastName`, you may have some documents which don't have an entry for `middleName`. If you pass `middleName` as an input to a skill in the pipeline, then it's expected that this skill input may be missing some of the time. You will need to evaluate your data and scenario to determine whether or not any action is required as a result of this warning.

If you want to provide a default value in case of missing input, you can use the Conditional skill to generate a default value and then use the output of the Conditional skill as the skill input.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document",
    "inputs": [
        { "name": "condition", "source": "= $(/document/language) == null" },
        { "name": "whenTrue", "source": "= 'en'" },
        { "name": "whenFalse", "source": "= $(/document/language)" }
    ],
    "outputs": [ { "name": "output", "targetName": "languageWithDefault" } ]
}
```

Reason	Details/Example	Resolution
Skill input is the wrong type	"Required skill input was not of the expected type `String`. Name: `text`, Source: `/document/merged_content`." "Required skill input was not of the expected format. Name: `text`, Source: `/document/merged_content`." "Cannot iterate over non-array `/document/normalized_images/0/imageCelebrities/0/detail/celebrities`." "Unable to select `0` in non-array `/document/normalized_images/0/imageCelebrities/0/detail/celebrities`"	Certain skills expect inputs of particular types, for example Sentiment skill expects `text` to be a string. If the input specifies a non-string value, then the skill doesn't execute and generates no outputs. Ensure your data set has input values uniform in type, or use a Custom Web API skill to preprocess the input. If you're iterating the skill over an array, check the skill context and input have `*` in the correct positions. Usually both the context and input source should end with `*` for arrays.

Reason	Details/Example	Resolution
Skill input is missing	`Required skill input is missing. Name: text, Source: /document/merged_content` `Missing value /document/normalized_images/0/imageTags.` `Unable to select 0 in array /document/pages of length 0.`	If this warning occurs for all documents, there could be a typo in the input paths. Check the property name casing. Check for an extra or missing `*` in the path. Verify that the documents from the data source provide the required inputs.
Skill language code input is invalid	Skill input `languageCode` has the following language codes `X,Y,Z`, at least one of which is invalid.	See more details below.

`Warning: Skill input 'languageCode' has the following language codes 'X,Y,Z', at least one of which is invalid.`

One or more of the values passed into the optional `languageCode` input of a downstream skill isn't supported. This can occur if you're passing the output of the LanguageDetectionSkill to subsequent skills, and the output consists of more languages than are supported in those downstream skills.

Note that you may also get a warning similar to this one if an invalid `countryHint` input gets passed to the LanguageDetectionSkill. If that happens, validate that the field you're using from your data source for that input contains valid ISO 3166-1 alpha-2 two letter country codes. If some are valid and some are invalid, continue with the following guidance but replace `languageCode` with `countryHint` and `defaultLanguageCode` with `defaultCountryHint` to match your use case.

If you know that your data set is all in one language, you should remove the LanguageDetectionSkill and the `languageCode` skill input and use the `defaultLanguageCode` skill parameter for that skill instead, assuming the language is supported for that skill.

If you know that your data set contains multiple languages and thus you need the LanguageDetectionSkill and `languageCode` input, consider adding a ConditionalSkill to filter out the text with languages that are not supported before passing in the text to the downstream skill. Here's an example of what this might look like for the EntityRecognitionSkill:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document",
    "inputs": [
        { "name": "condition", "source": "= $(/document/language) == 'de' || $(/document/language) == 'en'
|| $(/document/language) == 'es' || $(/document/language) == 'fr' || $(/document/language) == 'it'" },
        { "name": "whenTrue", "source": "/document/content" },
        { "name": "whenFalse", "source": "= null" }
    ],
    "outputs": [ { "name": "output", "targetName": "supportedByEntityRecognitionSkill" } ]
}
```

Here are some references for the currently supported languages for each of the skills that may produce this error message:

- EntityRecognitionSkill supported languages
- EntityLinkingSkill supported languages
- KeyPhraseExtractionSkill supported languages
- LanguageDetectionSkill supported languages
- PIIDetectionSkill supported languages
- SentimentSkill supported languages
- Translator supported languages
- Text SplitSkill supported languages: `da, de, en, es, fi, fr, it, ko, pt`

`Warning: Skill input was truncated`

Cognitive skills limit the length of text that can be analyzed at one time. If the text input exceeds the limit, the text is truncated before it's enriched. The skill executes, but not over all of your data.

In the example `LanguageDetectionSkill` below, the `'text'` input field might trigger this warning if the input is over the character limit. Input limits can be found in the skills reference documentation.

JSON

```
{
   "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
   "inputs": [
     {
       "name": "text",
       "source": "/document/text"
     }
   ],
   "outputs": [...]
 }
```

If you want to ensure that all text is analyzed, consider using the Split skill.

`Warning: Web API skill response contains warnings`

The indexer ran the skill in the skillset, but the response from the Web API request indicates there are warnings. Review the warnings to understand how your data is impacted and whether further action is required.

`Warning: The current indexer configuration does not support incremental progress`

This warning only occurs for Azure Cosmos DB data sources.

Incremental progress during indexing ensures that if indexer execution is interrupted by transient failures or execution time limit, the indexer can pick up where it left off next time it runs, instead of having to re-index the entire collection from scratch. This is especially important when indexing large collections.

The ability to resume an unfinished indexing job is predicated on having documents ordered by the `_ts` column. The indexer uses the timestamp to determine which document to pick up next. If the `_ts` column is missing or if the indexer can't determine if a custom query is ordered by it, the indexer starts at beginning and you'll see this warning.

It's possible to override this behavior, enabling incremental progress and suppressing this warning by using the `assumeOrderByHighWaterMarkColumn` configuration property.

For more information, see Incremental progress and custom queries.

`Warning: Some data was lost during projection. Row 'X' in table 'Y' has string property 'Z' which was too long.`

The Table Storage service has limits on how large entity properties can be. Strings can have 32,000 characters or less. If a row with a string property longer than 32,000 characters is being projected, only the first 32,000 characters are preserved. To work around this issue, avoid projecting rows with string properties longer than 32,000 characters.

`Warning: Truncated extracted text to X characters`

Indexers limit how much text can be extracted from any one document. This limit depends on the pricing tier: 32,000 characters for Free tier, 64,000 for Basic, 4 million for Standard, 8 million for Standard S2, and 16 million for Standard S3.

Text that was truncated won't be indexed. To avoid this warning, try breaking apart documents with large amounts of text into multiple, smaller documents.

For more information, see Indexer limits.

`Warning: Could not map output field 'X' to search index`

Output field mappings that reference non-existent/null data will produce warnings for each document and result in an empty index field. To work around this issue, double-check your output field-mapping source paths for possible typos, or set a default value using the Conditional skill. See Output field mapping for details.

Reason	Details/Example	Resolution
Cannot iterate over non-array	"Cannot iterate over non-array `/document/normalized_images/0/imageCelebrities/0/detail/celebrities`."	This error occurs when the output isn't an array. If you think the output should be an array, check the indicated output source field path for errors. For example, you might have a missing or extra `*` in the source field name. It's also possible that the input to this skill is null, resulting in an empty array. Find similar details in Skill Input was Invalid section.
Unable to select `0` in non-array	"Unable to select `0` in non-array `/document/pages`."	This could happen if the skills output doesn't produce an array and the output source field name has array index or `*` in its path. Double check the paths provided in the output source field names and the field value for the indicated field name. Find similar details in Skill Input was Invalid section.

`Warning: The data change detection policy is configured to use key column 'X'`

Data change detection policies have specific requirements for the columns they use to detect change. One of these requirements is that this column is updated every time the source item is changed. Another requirement is that the new value for this column is greater than the previous value. Key columns don't fulfill this requirement because they don't change on every update. To work around this issue, select a different column for the change detection policy.

`Warning: Document text appears to be UTF-16 encoded, but is missing a byte order mark`

The indexer parsing modes need to know how text is encoded before parsing it. The two most common ways of encoding text are UTF-16 and UTF-8. UTF-8 is a variable-length encoding where each character is between 1 byte and 4 bytes long. UTF-16 is a fixed-length encoding where each character is 2 bytes long. UTF-16 has two different variants, "big endian" and "little endian". Text encoding is determined by a "byte order mark", a series of bytes before the text.

Encoding	Byte Order Mark
UTF-16 Big Endian	0xFE 0xFF
UTF-16 Little Endian	0xFF 0xFE
UTF-8	0xEF 0xBB 0xBF

If no byte order mark is present, the text is assumed to be encoded as UTF-8.

To work around this warning, determine what the text encoding for this blob is and add the appropriate byte order mark.

`Warning: Azure Cosmos DB collection 'X' has a Lazy indexing policy. Some data may be lost`

Collections with Lazy indexing policies can't be queried consistently, resulting in your indexer missing data. To work around this warning, change your indexing policy to Consistent.

`Warning: The document contains very long words (longer than 64 characters). These words may result in truncated and/or unreliable model predictions.`

This warning is passed from the Language service of Azure Cognitive Services. In some cases, it's safe to ignore this warning, such as when your document contains a long URL (which likely isn't a key phrase or driving sentiment, etc.). Be aware that when a word is longer than 64 characters, it will be truncated to 64 characters which can affect model predictions.

Search over Azure Blob Storage content

Article • 03/10/2023 • 7 minutes to read

Searching across the variety of content types stored in Azure Blob Storage can be a difficult problem to solve, but Azure Cognitive Search provides deep integration at the content layer, extracting and inferring textual information, which can then be queried in a search index.

In this article, review the basic workflow for extracting content and metadata from blobs and sending it to a search index in Azure Cognitive Search. The resulting index can be queried using full text search. Optionally, you can send processed blob content to a knowledge store for non-search scenarios.

> **Note**
>
> Already familiar with the workflow and composition? **Configure a blob indexer** is your next step.

What it means to add full text search to blob data

Azure Cognitive Search is a standalone search service that supports indexing and query workloads over user-defined indexes that contain your remote searchable content hosted in the cloud. Co-locating your searchable content with the query engine is necessary for performance, returning results at a speed users have come to expect from search queries.

Cognitive Search integrates with Azure Blob Storage at the indexing layer, importing your blob content as search documents that are indexed into *inverted indexes* and other query structures that support free-form text queries and filter expressions. Because your blob content is indexed into a search index, you can use the full range of query features in Azure Cognitive Search to find information in your blob content.

Inputs are your blobs, in a single container, in Azure Blob Storage. Blobs can be almost any kind of text data. If your blobs contain images, you can add AI enrichment to create and extract text from images.

Output is always an Azure Cognitive Search index, used for fast text search, retrieval, and exploration in client applications. In between is the indexing pipeline architecture itself. The pipeline is based on the *indexer* feature, discussed further on in this article.

Once the index is created and populated, it exists independently of your blob container, but you can rerun indexing operations to refresh your index based on changed documents. Timestamp information on individual blobs is used for change detection. You can opt for either scheduled execution or on-demand indexing as the refresh mechanism.

Resources used in a blob-search solution

You need Azure Cognitive Search, Azure Blob Storage, and a client. Cognitive Search is typically one of several components in a solution, where your application code issues query API requests and handles the response. You might also write application code to handle indexing, although for proof-of-concept testing and impromptu tasks, it's common to use the Azure portal as the search client.

Within Blob Storage, you'll need a container that provides source content. You can set file inclusion and exclusion criteria, and specify which parts of a blob are indexed in Cognitive Search.

You can start directly in your Storage Account portal page.

1. In the left navigation page under **Data management**, select **Azure search** to select or create a search service.

2. Follow the steps in the wizard to extract and optionally create searchable content from your blobs. The workflow is the **Import data** wizard.

3. Use Search explorer in the search portal page to query your content.

The wizard is the best place to start, but you'll discover more flexible options when you configure a blob indexer yourself. You can call the REST APIs using a tool like Postman. Tutorial: Index and search semi-structured data (JSON blobs) in Azure Cognitive Search walks you through the steps of calling the REST API in Postman.

How blobs are indexed

By default, most blobs are indexed as a single search document in the index, including blobs with structured content, such as JSON or CSV, which are indexed as a single chunk of text. However, for JSON or CSV documents that have an internal structure (delimiters), you can assign parsing modes to generate individual search documents for each line or element:

- Indexing JSON blobs
- Indexing CSV blobs

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding AI enrichment to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

> **Note**
>
> Azure Cognitive Search imposes **indexer limits** on how much text it extracts depending on the pricing tier. A warning will appear in the indexer status response if documents are truncated.

Use a Blob indexer for content extraction

An *indexer* is a data-source-aware subservice in Cognitive Search, equipped with internal logic for sampling data, reading and retrieving data and metadata, and serializing data from native formats into JSON documents for subsequent import.

Blobs in Azure Storage are indexed using the blob indexer. You can invoke this indexer by using the **Azure search** command in Azure Storage, the **Import data** wizard, a REST API, or the .NET SDK. In code, you use this indexer by setting the type, and by providing connection information that includes an Azure Storage account along with a blob container. You can subset your blobs by creating a virtual directory, which you can then pass as a parameter, or by filtering on a file type extension.

An indexer "cracks a document", opening a blob to inspect content. After connecting to the data source, it's the first step in the pipeline. For blob data, this is where PDF, Office docs, and other content types are detected. Document cracking with text extraction is no charge. If your blobs contain image content, images are ignored unless you add AI enrichment. Standard indexing applies only to text content.

The Blob indexer comes with configuration parameters and supports change tracking if the underlying data provides sufficient information. You can learn more about the core functionality in Blob indexer.

Supported access tiers

Blob storage access tiers include hot, cool, and archive. Only hot and cool can be accessed by indexers.

Supported content types

By running a Blob indexer over a container, you can extract text and metadata from the following content types with a single query:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

Controlling which blobs are indexed

You can control which blobs are indexed, and which are skipped, by the blob's file type or by setting properties on the blob themselves, causing the indexer to skip over them.

Include specific file extensions by setting `"indexedFileNameExtensions"` to a comma-separated list of file extensions (with a leading dot). Exclude specific file extensions by setting `"excludedFileNameExtensions"` to the extensions that should be skipped. If the same extension is in both lists, it will be excluded from indexing.

HTTP

```
PUT /indexers/[indexer name]?api-version=2020-06-30
{
    "parameters" : {
        "configuration" : {
            "indexedFileNameExtensions" : ".pdf, .docx",
            "excludedFileNameExtensions" : ".png, .jpeg"
        }
    }
}
```

Add "skip" metadata the blob

The indexer configuration parameters apply to all blobs in the container or folder. Sometimes, you want to control how *individual blobs* are indexed.

Add the following metadata properties and values to blobs in Blob Storage. When the indexer encounters this property, it will skip the blob or its content in the indexing run.

Property name	Property value	Explanation
"AzureSearch_Skip"	`"true"`	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
"AzureSearch_SkipContent"	`"true"`	This is equivalent to the `"dataToExtract" : "allMetadata"` setting described above scoped to a particular blob.

Indexing blob metadata

A common scenario that makes it easy to sort through blobs of any content type is to index both custom metadata and system properties for each blob. In this way, information for all blobs is indexed regardless of document type, stored in an index in your search service. Using your new index, you can then proceed to sort, filter, and facet across all Blob storage content.

> **Note**
>
> Blob Index tags are natively indexed by the Blob storage service and exposed for querying. If your blobs' key/value attributes require indexing and filtering capabilities, Blob Index tags should be leveraged instead of metadata.
>
> To learn more about Blob Index, see **Manage and find data on Azure Blob Storage with Blob Index**.

Search blob content in a search index

The output of an indexer is a search index, used for interactive exploration using free text and filtered queries in a client app. For initial exploration and verification of content, we recommend starting with Search Explorer in the portal to examine document structure. In Search explorer, you can use:

- Simple query syntax
- Full query syntax
- Filter expression syntax

A more permanent solution is to gather query inputs and present the response as search results in a client application. The following C# tutorial explains how to build a search application: Add search to an ASP.NET Core (MVC) application.

Next steps

- Upload, download, and list blobs with the Azure portal (Azure Blob storage)
- Set up a blob indexer (Azure Cognitive Search)

Additional resources

Documentation

Manage and find Azure Blob data with blob index tags

Learn how to use blob index tags to categorize, manage, and query for blob objects.

Tutorial: Index semi-structured data in JSON blobs - Azure Cognitive Search

Learn how to index and search semi-structured Azure JSON blobs using Azure Cognitive Search REST APIs and Postman.

Query Blob Contents (REST API) - Azure Storage

The Query Blob Contents operation applies a simple Structured Query Language (SQL) statement on a blob's contents and returns only the queried subset of the data.

Connect to Azure Storage - Azure Cognitive Search

Learn how to set up an indexer connection to an Azure Storage account using a managed identity

Azure Blob indexer - Azure Cognitive Search

Set up an Azure Blob indexer to automate indexing of blob content for full text search operations and knowledge mining in Azure Cognitive Search.

Azure Files indexer (preview) - Azure Cognitive Search

Set up an Azure Files indexer to automate indexing of file shares in Azure Cognitive Search.

Use blob index tags to manage and find data on Azure Blob Storage - Azure Storage

See examples of how to use blob index tags to categorize, manage, and query for blob objects.

Search over JSON blobs - Azure Cognitive Search

Crawl Azure JSON blobs for text content using the Azure Cognitive Search Blob indexer. Indexers automate data ingestion for selected data sources like Azure Blob Storage.

Show 5 more

Training

Module

Organize Azure storage blobs with properties and metadata - Training

Find out about Azure storage blob properties and metadata, how to organize blob data, and create an app with .NET to retrieve metadata.

Index data from Azure Data Lake Storage Gen2

Article • 03/23/2023 • 13 minutes to read

In this article, learn how to configure an **indexer** that imports content from Azure Data Lake Storage (ADLS) Gen2 and makes it searchable in Azure Cognitive Search. Inputs to the indexer are your blobs, in a single container. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements **Create an indexer** with information that's specific to indexing from ADLS Gen2. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

For a code sample in C#, see Index Data Lake Gen2 using Azure AD on GitHub.

Prerequisites

- ADLS Gen2 with hierarchical namespace enabled. ADLS Gen2 is available through Azure Storage. When setting up a storage account, you have the option of enabling hierarchical namespace, organizing files into a hierarchy of directories and nested subdirectories. By enabling a hierarchical namespace, you enable ADLS Gen2.
- Access tiers for ADLS Gen2 include hot, cool, and archive. Only hot and cool can be accessed by search indexers.
- Blobs containing text. If you have binary data, you can include AI enrichment for image analysis. Blob content can't exceed the indexer limits for your search service tier.
- Read permissions on Azure Storage. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles instead, make sure the search service managed identity has **Storage Blob Data Reader** permissions.
- Use a REST client, such as Postman app, if you want to formulate REST calls similar to the ones shown in this article.

Note

ADLS Gen2 implements an **access control model** that supports both Azure role-based access control (Azure RBAC) and POSIX-like access control lists (ACLs) at the blob level. Azure Cognitive Search does not support document-level permissions. All users have the same level of access to all searchable and retrievable content in the index. If document-level permissions are an application requirement, consider **security trimming** as a potential solution.

Supported document formats

The ADLS Gen2 indexer can extract text from the following document formats:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

Determine which blobs to index

Before you set up indexing, review your source data to determine whether any changes should be made up front. An indexer can index content from one container at a time. By default, all blobs in the container are processed. You have several options for more selective processing:

- Place blobs in a virtual folder. An indexer data source definition includes a "query" parameter that can take a virtual folder. If you specify a virtual folder, only those blobs in the folder are indexed.

- Include or exclude blobs by file type. The supported document formats list can help you determine which blobs to exclude. For example, you might want to

exclude image or audio files that don't provide searchable text. This capability is controlled through configuration settings in the indexer.

- Include or exclude arbitrary blobs. If you want to skip a specific blob for whatever reason, you can add the following metadata properties and values to blobs in Blob Storage. When an indexer encounters this property, it skips the blob or its content in the indexing run.

Property name	Property value	Explanation
"AzureSearch_Skip"	`"true"`	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
"AzureSearch_SkipContent"	`"true"`	Skips content and extracts just the metadata. this is equivalent to the `"dataToExtract" : "allMetadata"` setting described in configuration settings , just scoped to a particular blob.

If you don't set up inclusion or exclusion criteria, the indexer will report an ineligible blob as an error and move on. If enough errors occur, processing might stop. You can specify error tolerance in the indexer configuration settings.

An indexer typically creates one search document per blob, where the text content and metadata are captured as searchable fields in an index. If blobs are whole files, you can potentially parse them into multiple search documents. For example, you can parse rows in a CSV file to create one search document per row.

Indexing blob metadata

Blob metadata can also be indexed, and that's helpful if you think any of the standard or custom metadata properties will be useful in filters and queries.

User-specified metadata properties are extracted verbatim. To receive the values, you must define field in the search index of type `Edm.String`, with same name as the metadata key of the blob. For example, if a blob has a metadata key of `Sensitivity` with value `High`, you should define a field named `Sensitivity` in your search index and it will be populated with the value `High`.

Standard blob metadata properties can be extracted into similarly named and typed fields, as listed below. The blob indexer automatically creates internal field mappings for

these blob metadata properties, converting the original hyphenated name ("metadata-storage-name") to an underscored equivalent name ("metadata_storage_name").

You still have to add the underscored fields to the index definition, but you can omit field mappings because the indexer will make the association automatically.

- **metadata_storage_name** (`Edm.String`) - the file name of the blob. For example, if you have a blob /my-container/my-folder/subfolder/resume.pdf, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the blob, including the storage account. For example, `https://myaccount.blob.core.windows.net/my-container/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the blob. For example, `application/octet-stream`.
- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the blob. Azure Cognitive Search uses this timestamp to identify changed blobs, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - blob size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the blob content, if available.
- **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by custom skills to get access to the blob. This token shouldn't be stored for later use as it might expire.

Lastly, any metadata properties specific to the document format of the blobs you're indexing can also be represented in the index schema. For more information about content-specific metadata, see Content metadata properties.

It's important to point out that you don't need to define fields for all of the above properties in your search index - just capture the properties you need for your application.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create or update a data source to set its definition:

```JSON
{
    "name" : "my-adlsgen2-datasource",
    "type" : "adlsgen2",
    "credentials" : { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=
<account key>;" },
    "container" : { "name" : "my-container", "query" : "<optional-
virtual-directory-name>" }
}
```

2. Set "type" to `"adlsgen2"` (required).

3. Set `"credentials"` to an Azure Storage connection string. The next section describes the supported formats.

4. Set `"container"` to the blob container, and use "query" to specify any subfolders.

A data source definition can also include soft deletion policies, if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a blob container using the following connections.

Full access storage account connection string
`{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }`
You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }`
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity.

Storage account shared access signature** (SAS) connection string
`{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=rl;" }`
The SAS should have the list and read permissions on containers and objects (blobs in this case).

Container shared access signature
`{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=rl;" }`
The SAS should have the list and read permissions on the container. For more information, see Using Shared Access Signatures.

> **Note**
>
> If you use SAS credentials, you will need to update the data source credentials periodically with renewed signatures to prevent their expiration. If SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Add search fields to an index

In a search index, add fields to accept the content and metadata of your Azure blobs.

1. Create or update an index to define search fields that will store blob content and metadata:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
{
    "name" : "my-search-index",
    "fields": [
        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false },
        { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true  },
        { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": true, "sortable": true  },
```

```
        { "name": "metadata_storage_content_type", "type":
"Edm.String", "searchable": false, "filterable": true, "sortable": true
},
    ]
  }
}
```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties.

   - `metadata_storage_path` (default) full path to the object or file. The key field ("ID" in this example) will be populated with values from metadata_storage_path because it's the default.

   - `metadata_storage_name`, usable only if names are unique. If you want this field as the key, move `"key": true` to this field definition.

   - A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

   Metadata properties often include characters, such as `/` and `-`, that are invalid for document keys. Because the indexer has a "base64EncodeKeys" property (true by default), it automatically encodes the metadata property, with no configuration or field mapping required.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.

4. Add fields for standard metadata properties. The indexer can read custom metadata properties, standard metadata properties, and content-specific metadata properties.

Configure and run the ADLS Gen2 indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors. You can also specify which parts of a blob to index.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
{
  "name" : "my-adlsgen2-indexer,
  "dataSourceName" : "my-adlsgen2-datasource",
  "targetIndexName" : "my-search-index",
  "parameters": {
      "batchSize": null,
      "maxFailedItems": null,
      "maxFailedItemsPerBatch": null,
      "base64EncodeKeys": null,
      "configuration:" {
          "indexedFileNameExtensions" : ".pdf,.docx",
          "excludedFileNameExtensions" : ".png,.jpeg",
          "dataToExtract": "contentAndMetadata",
          "parsingMode": "default"
      }
  },
  "schedule" : { },
  "fieldMappings" : [ ]
}
```

2. Set "batchSize" if the default (10 documents) is either under utilizing or overwhelming available resources. Default batch sizes are data source specific. Blob indexing sets batch size at 10 documents in recognition of the larger average document size.

3. Under "configuration", control which blobs are indexed based on file type, or leave unspecified to retrieve all blobs.

   For `"indexedFileNameExtensions"`, provide a comma-separated list of file extensions (with a leading dot). Do the same for `"excludedFileNameExtensions"` to indicate which extensions should be skipped. If the same extension is in both lists, it will be excluded from indexing.

4. Under "configuration", set "dataToExtract" to control which parts of the blobs are indexed:

   - "contentAndMetadata" specifies that all metadata and textual content extracted from the blob are indexed. This is the default value.
   - "storageMetadata" specifies that only the standard blob properties and user-specified metadata are indexed.

- "allMetadata" specifies that standard blob properties and any metadata for found content types are extracted from the blob content and indexed.

5. Under "configuration", set "parsingMode" if blobs should be mapped to multiple search documents, or if they consist of plain text, JSON documents, or CSV files.

6. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

   In blob indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens `-` with underscores in the search index.

7. See Create an indexer for more information about other properties. For the full list of parameter descriptions, see Blob configuration parameters in the REST API.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
```

```
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2022-02-21T00:23:24.957Z",
                "endTime":"2022-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Handle errors

Errors that commonly occur during indexing include unsupported content types, missing content, or oversized blobs.

By default, the blob indexer stops as soon as it encounters a blob with an unsupported content type (for example, an audio file). You could use the "excludedFileNameExtensions" parameter to skip certain content types. However, you might want to indexing to proceed even if errors occur, and then debug individual documents later. For more information about indexer errors, see Indexer troubleshooting guidance and Indexer errors and warnings.

There are five indexer properties that control the indexer's response when errors occur.

HTTP

```
PUT /indexers/[indexer name]?api-version=2020-06-30
{
  "parameters" : {
    "maxFailedItems" : 10,
    "maxFailedItemsPerBatch" : 10,
    "configuration" : {
        "failOnUnsupportedContentType" : false,
        "failOnUnprocessableDocument" : false,
        "indexStorageMetadataOnlyForOversizedDocuments": false
```

```
    }
  }
```

Parameter	Valid values	Description
"maxFailedItems"	-1, null or 0, positive integer	Continue indexing if errors happen at any point of processing, either while parsing blobs or while adding documents to an index. Set these properties to the number of acceptable failures. A value of `-1` allows processing no matter how many errors occur. Otherwise, the value is a positive integer.
"maxFailedItemsPerBatch"	-1, null or 0, positive integer	Same as above, but used for batch indexing.
"failOnUnsupportedContentType"	true or false	If the indexer is unable to determine the content type, specify whether to continue or fail the job.
"failOnUnprocessableDocument"	true or false	If the indexer is unable to process a document of an otherwise supported content type, specify whether to continue or fail the job.
"indexStorageMetadataOnlyForOversizedDocuments"	true or false	Oversized blobs are treated as errors by default. If you set this parameter to true, the indexer will try to index its metadata even if the content cannot be indexed. For limits on blob size, see service Limits.

Next steps

You can now run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure Storage:

- Change detection and deletion detection

- Index large data sets
- C# Sample: Index Data Lake Gen2 using Azure AD

Index data from Azure Blob Storage

Article • 05/18/2023

In this article, learn how to configure an **indexer** that imports content from Azure Blob Storage and makes it searchable in Azure Cognitive Search. Inputs to the indexer are your blobs, in a single container. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements **Create an indexer** with information that's specific to Blob Storage. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Blob indexers are frequently used for both AI enrichment and text-based processing. This article focuses on indexers for text-based indexing, where just the textual content and metadata are ingested for full text search scenarios.

Prerequisites

- Azure Blob Storage, Standard performance (general-purpose v2).
- Access tiers for Blob Storage include Hot, Cool, and Archive. Only Hot and Cool can be accessed by search indexers.
- Blobs providing text content and metadata. If blobs contain binary content or unstructured text, consider adding AI enrichment for image and natural language processing. Blob content can't exceed the indexer limits for your search service tier.
- A supported network configuration and data access. At a minimum, you'll need read permissions in Azure Storage. A storage connection string that includes an access key will give you read access to storage content. If instead you're using Azure AD logins and roles, make sure the search service's managed identity has **Storage Blob Data Reader** permissions.

  By default, both search and storage accept requests from public IP addresses. If network security isn't an immediate concern, you can index blob data using just the connection string and read permissions. When you're ready to add network protections, see Indexer access to content protected by Azure network security features for guidance about data access.
- Use a REST client, such as Postman app, if you want to formulate REST calls similar to the ones shown in this article.

Supported document formats

The blob indexer can extract text from the following document formats:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

Determine which blobs to index

Before you set up indexing, review your source data to determine whether any changes should be made up front. An indexer can index content from one container at a time. By default, all blobs in the container are processed. You have several options for more selective processing:

- Place blobs in a virtual folder. An indexer data source definition includes a "query" parameter that can take a virtual folder. If you specify a virtual folder, only those blobs in the folder are indexed.

- Include or exclude blobs by file type. The supported document formats list can help you determine which blobs to exclude. For example, you might want to exclude image or audio files that don't provide searchable text. This capability is controlled through configuration settings in the indexer.

- Include or exclude arbitrary blobs. If you want to skip a specific blob for whatever reason, you can add the following metadata properties and values to blobs in Blob Storage. When an indexer encounters this property, it skips the blob or its content in the indexing run.

Property name	Property value	Explanation
"AzureSearch_Skip"	`"true"`	Instructs the blob indexer to completely skip the blob. Neither metadata nor content extraction is attempted. This is useful when a particular blob fails repeatedly and interrupts the indexing process.
"AzureSearch_SkipContent"	`"true"`	Skips content and extracts just the metadata. this is equivalent to the `"dataToExtract" : "allMetadata"` setting described in configuration settings , just scoped to a particular blob.

If you don't set up inclusion or exclusion criteria, the indexer will report an ineligible blob as an error and move on. If enough errors occur, processing might stop. You can specify error tolerance in the indexer configuration settings.

An indexer typically creates one search document per blob, where the text content and metadata are captured as searchable fields in an index. If blobs are whole files, you can potentially parse them into multiple search documents. For example, you can parse rows in a CSV file to create one search document per row.

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding AI enrichment to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

Indexing blob metadata

Blob metadata can also be indexed, and that's helpful if you think any of the standard or custom metadata properties will be useful in filters and queries.

User-specified metadata properties are extracted verbatim. To receive the values, you must define field in the search index of type `Edm.String`, with same name as the metadata key of the blob. For example, if a blob has a metadata key of `Sensitivity` with value `High`, you should define a field named `Sensitivity` in your search index and it will be populated with the value `High`.

Standard blob metadata properties can be extracted into similarly named and typed fields, as listed below. The blob indexer automatically creates internal field mappings for these blob metadata properties, converting the original hyphenated name ("metadata-storage-name") to an underscored equivalent name ("metadata_storage_name").

You still have to add the underscored fields to the index definition, but you can omit field mappings because the indexer will make the association automatically.

- **metadata_storage_name** (`Edm.String`) - the file name of the blob. For example, if you have a blob /my-container/my-folder/subfolder/resume.pdf, the value of this field is `resume.pdf`.
- **metadata_storage_path** (`Edm.String`) - the full URI of the blob, including the storage account. For example, `https://myaccount.blob.core.windows.net/my-container/my-folder/subfolder/resume.pdf`
- **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the blob. For example, `application/octet-stream`.
- **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the blob. Azure Cognitive Search uses this timestamp to identify changed blobs, to avoid reindexing everything after the initial indexing.
- **metadata_storage_size** (`Edm.Int64`) - blob size in bytes.
- **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the blob content, if available.
- **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by custom skills to get access to the blob. This token shouldn't be stored for later use as it might expire.

Lastly, any metadata properties specific to the document format of the blobs you're indexing can also be represented in the index schema. For more information about content-specific metadata, see Content metadata properties.

It's important to point out that you don't need to define fields for all of the above properties in your search index - just capture the properties you need for your application.

Currently, indexing blob index tags is not supported by this indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create or update a data source to set its definition:

```JSON
{
    "name" : "my-blob-datasource",
    "type" : "azureblob",
    "credentials" : { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=
<account key>;" },
    "container" : { "name" : "my-container", "query" : "<optional-
virtual-directory-name>" }
}
```

2. Set "type" to `"azureblob"` (required).

3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.

4. Set "container" to the blob container, and use "query" to specify any subfolders.

A data source definition can also include soft deletion policies, if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a blob container using the following connections.

Full access storage account connection string
`{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }`
You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }`

Managed identity connection string
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity.

Storage account shared access signature** (SAS) connection string
`{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=rl;" }`
The SAS should have the list and read permissions on containers and objects (blobs in this case).

Container shared access signature
`{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=rl;" }`
The SAS should have the list and read permissions on the container. For more information, see Using Shared Access Signatures.

> **Note**
>
> If you use SAS credentials, you will need to update the data source credentials periodically with renewed signatures to prevent their expiration. If SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Add search fields to an index

In a search index, add fields to accept the content and metadata of your Azure blobs.

1. Create or update an index to define search fields that will store blob content and metadata:

```http
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
{
    "name" : "my-search-index",
    "fields": [
        { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
```

```
        { "name": "content", "type": "Edm.String", "searchable": true,
"filterable": false },
        { "name": "metadata_storage_name", "type": "Edm.String",
"searchable": false, "filterable": true, "sortable": true  },
        { "name": "metadata_storage_size", "type": "Edm.Int64",
"searchable": false, "filterable": true, "sortable": true  },
        { "name": "metadata_storage_content_type", "type":
"Edm.String", "searchable": false, "filterable": true, "sortable": true
},
    ]
  }
}
```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties.

   - `metadata_storage_path` (default) full path to the object or file. The key field ("ID" in this example) will be populated with values from metadata_storage_path because it's the default.

   - `metadata_storage_name`, usable only if names are unique. If you want this field as the key, move `"key": true` to this field definition.

   - A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

   Metadata properties often include characters, such as `/` and `-`, that are invalid for document keys. Because the indexer has a "base64EncodeKeys" property (true by default), it automatically encodes the metadata property, with no configuration or field mapping required.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.

4. Add fields for standard metadata properties. The indexer can read custom metadata properties, standard metadata properties, and content-specific metadata properties.

Configure and run the blob indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors. You can also specify which parts of a blob to index.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

   HTTP

   ```
   POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
   {
     "name" : "my-blob-indexer",
     "dataSourceName" : "my-blob-datasource",
     "targetIndexName" : "my-search-index",
     "parameters": {
         "batchSize": null,
         "maxFailedItems": null,
         "maxFailedItemsPerBatch": null,
         "base64EncodeKeys": null,
         "configuration": {
             "indexedFileNameExtensions" : ".pdf,.docx",
             "excludedFileNameExtensions" : ".png,.jpeg",
             "dataToExtract": "contentAndMetadata",
             "parsingMode": "default"
         }
     },
     "schedule" : { },
     "fieldMappings" : [ ]
   }
   ```

2. Set `batchSize` if the default (10 documents) is either underutilizing or overwhelming available resources. Default batch sizes are data source specific. Blob indexing sets batch size at 10 documents in recognition of the larger average document size.

3. Under "configuration", control which blobs are indexed based on file type, or leave unspecified to retrieve all blobs.

   For `"indexedFileNameExtensions"`, provide a comma-separated list of file extensions (with a leading dot). Do the same for `"excludedFileNameExtensions"` to indicate which extensions should be skipped. If the same extension is in both lists, it will be excluded from indexing.

4. Under "configuration", set "dataToExtract" to control which parts of the blobs are indexed:

- "contentAndMetadata" specifies that all metadata and textual content extracted from the blob are indexed. This is the default value.
- "storageMetadata" specifies that only the standard blob properties and user-specified metadata are indexed.
- "allMetadata" specifies that standard blob properties and any metadata for found content types are extracted from the blob content and indexed.

5. Under "configuration", set "parsingMode" if blobs should be mapped to multiple search documents, or if they consist of plain text, JSON documents, or CSV files.
6. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

   In blob indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens `-` with underscores in the search index.
7. See Create an indexer for more information about other properties. For the full list of parameter descriptions, see Blob configuration parameters in the REST API.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
```

```
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2022-02-21T00:23:24.957Z",
                "endTime":"2022-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Handle errors

Errors that commonly occur during indexing include unsupported content types, missing content, or oversized blobs.

By default, the blob indexer stops as soon as it encounters a blob with an unsupported content type (for example, an audio file). You could use the "excludedFileNameExtensions" parameter to skip certain content types. However, you might want to indexing to proceed even if errors occur, and then debug individual documents later. For more information about indexer errors, see Indexer troubleshooting guidance and Indexer errors and warnings.

There are five indexer properties that control the indexer's response when errors occur.

HTTP

```
PUT /indexers/[indexer name]?api-version=2020-06-30
{
```

```
  "parameters" : {
    "maxFailedItems" : 10,
    "maxFailedItemsPerBatch" : 10,
    "configuration" : {
        "failOnUnsupportedContentType" : false,
        "failOnUnprocessableDocument" : false,
        "indexStorageMetadataOnlyForOversizedDocuments": false
  }
}
```

Parameter	Valid values	Description
"maxFailedItems"	-1, null or 0, positive integer	Continue indexing if errors happen at any point of processing, either while parsing blobs or while adding documents to an index. Set these properties to the number of acceptable failures. A value of `-1` allows processing no matter how many errors occur. Otherwise, the value is a positive integer.
"maxFailedItemsPerBatch"	-1, null or 0, positive integer	Same as above, but used for batch indexing.
"failOnUnsupportedContentType"	true or false	If the indexer is unable to determine the content type, specify whether to continue or fail the job.
"failOnUnprocessableDocument"	true or false	If the indexer is unable to process a document of an otherwise supported content type, specify whether to continue or fail the job.
"indexStorageMetadataOnlyForOversizedDocuments"	true or false	Oversized blobs are treated as errors by default. If you set this parameter to true, the indexer will try to index its metadata even if the content can't be indexed. For limits on blob size, see service Limits.

Next steps

You can now control how you run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure Storage:

- Change detection and deletion detection
- Index large data sets
- Indexer access to content protected by Azure network security features

Index data from Azure Files

Article • 03/24/2023 • 8 minutes to read

> **Important**
>
> Azure Files indexer is currently in public preview under **Supplemental Terms of Use**. Use a **preview REST API (2020-06-30-preview or later)** to create the indexer data source.

In this article, learn how to configure an **indexer** that imports content from Azure Files and makes it searchable in Azure Cognitive Search. Inputs to the indexer are your files in a single share. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements **Create an indexer** with information that's specific to indexing files in Azure Storage. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Prerequisites

- Azure Files, Transaction Optimized tier.
- An SMB file share providing the source content. NFS shares are not supported.
- Files containing text. If you have binary data, you can include AI enrichment for image analysis.
- Read permissions on Azure Storage. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles instead, make sure the search service managed identity has **Data and Reader** permissions.
- Use a REST client, such as Postman app, if you want to formulate REST calls similar to the ones shown in this article.

Supported document formats

The Azure Files indexer can extract text from the following document formats:

- CSV (see Indexing CSV blobs)
- EML

- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

How Azure Files are indexed

By default, most files are indexed as a single search document in the index, including files with structured content, such as JSON or CSV, which are indexed as a single chunk of text.

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or an .MSG file with attachments) is also indexed as a single document. For example, all images extracted from the attachments of an .MSG file will be returned in the normalized_images field. If you have images, consider adding AI enrichment to get more search utility from that content.

Textual content of a document is extracted into a string field named "content". You can also extract standard and user-defined metadata.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create or update a data source to set its definition, using a preview API version 2020-06-30-Preview or 2021-04-30-Preview for "type": `"azurefile"`.

   JSON

   ```
   {
       "name" : "my-file-datasource",
   ```

```
    "type" : "azurefile",
    "credentials" : { "connectionString" :
"DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=
<account key>;" },
    "container" : { "name" : "my-file-share", "query" : "<optional-
directory-name>" }
}
```

2. Set "type" to `"azurefile"` (required).

3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.

4. Set "container" to the root file share, and use "query" to specify any subfolders.

A data source definition can also include soft deletion policies, if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a file share using the following connections.

Full access storage account connection string
`{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }`
You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }`
This connection string does not require an account key, but you must have previously configured a search service to connect using a managed identity.

Storage account shared access signature (SAS) connection string**
`{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=rl;" }`
The SAS should have the list and read permissions on containers and objects (blobs in this case).

Container shared access signature
`{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=rl;" }`
The SAS should have the list and read permissions on the container. For more information, see Using Shared Access Signatures.

> **Note**
>
> If you use SAS credentials, you will need to update the data source credentials periodically with renewed signatures to prevent their expiration. If SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Add search fields to an index

In the search index, add fields to accept the content and metadata of your Azure files.

1. Create or update an index to define search fields that will store file content and metadata:

```http
POST /indexes?api-version=2020-06-30
{
  "name" : "my-search-index",
  "fields": [
      { "name": "ID", "type": "Edm.String", "key": true, "searchable": false },
      { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false },
      { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true  },
      { "name": "metadata_storage_path", "type": "Edm.String", "searchable": false, "filterable": true, "sortable": true },
      { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": true, "sortable": true  },
      { "name": "metadata_storage_content_type", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true }
  ]
}
```

2. Create a document key field ("key": true). For blob content, the best candidates are metadata properties. Metadata properties often include characters, such as `/` and

`-`, that are invalid for document keys. Because the indexer has a "base64EncodeKeys" property (true by default), it automatically encodes the metadata property, with no configuration or field mapping required.

- **`metadata_storage_path`** (default) full path to the object or file
- **`metadata_storage_name`** usable only if names are unique
- A custom metadata property that you add to blobs. This option requires that your blob upload process adds that metadata property to all blobs. Since the key is a required property, any blobs that are missing a value will fail to be indexed. If you use a custom metadata property as a key, avoid making changes to that property. Indexers will add duplicate documents for the same blob if the key property changes.

3. Add a "content" field to store extracted text from each file through the blob's "content" property. You aren't required to use this name, but doing so lets you take advantage of implicit field mappings.

4. Add fields for standard metadata properties. In file indexing, the standard metadata properties are the same as blob metadata properties. The file indexer automatically creates internal field mappings for these properties that converts hyphenated property names to underscored property names. You still have to add the fields you want to use the index definition, but you can omit creating field mappings in the data source.

   - **metadata_storage_name** (`Edm.String`) - the file name. For example, if you have a file /my-share/my-folder/subfolder/resume.pdf, the value of this field is `resume.pdf`.
   - **metadata_storage_path** (`Edm.String`) - the full URI of the file, including the storage account. For example, `https://myaccount.file.core.windows.net/my-share/my-folder/subfolder/resume.pdf`
   - **metadata_storage_content_type** (`Edm.String`) - content type as specified by the code you used to upload the file. For example, `application/octet-stream`.
   - **metadata_storage_last_modified** (`Edm.DateTimeOffset`) - last modified timestamp for the file. Azure Cognitive Search uses this timestamp to identify changed files, to avoid reindexing everything after the initial indexing.
   - **metadata_storage_size** (`Edm.Int64`) - file size in bytes.
   - **metadata_storage_content_md5** (`Edm.String`) - MD5 hash of the file content, if available.
   - **metadata_storage_sas_token** (`Edm.String`) - A temporary SAS token that can be used by custom skills to get access to the file. This token shouldn't be

stored for later use as it might expire.

Configure and run the file indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

   HTTP

   ```
   POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
   {
     "name" : "my-file-indexer",
     "dataSourceName" : "my-file-datasource",
     "targetIndexName" : "my-search-index",
     "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "maxFailedItemsPerBatch": null,
        "base64EncodeKeys": null,
        "configuration": {
           "indexedFileNameExtensions" : ".pdf,.docx",
           "excludedFileNameExtensions" : ".png,.jpeg"
       }
     },
     "schedule" : { },
     "fieldMappings" : [ ]
   }
   ```

2. In the optional "configuration" section, provide any inclusion or exclusion criteria. If left unspecified, all files in the file share are retrieved.

   If both `indexedFileNameExtensions` and `excludedFileNameExtensions` parameters are present, Azure Cognitive Search first looks at `indexedFileNameExtensions`, then at `excludedFileNameExtensions`. If the same file extension is present in both lists, it will be excluded from indexing.

3. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

   In file indexing, you can often omit field mappings because the indexer has built-in support for mapping the "content" and metadata properties to similarly named

and typed fields in an index. For metadata properties, the indexer will automatically replace hyphens `-` with underscores in the search index.

4. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2022-02-21T00:23:24.957Z",
                "endTime":"2022-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
```

```
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Next steps

You can now run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure Storage:

- Change detection and deletion detection
- Index large data sets

Index data from Azure Table Storage

Article • 03/24/2023 • 7 minutes to read

In this article, learn how to configure an **indexer** that imports content from Azure Table Storage and makes it searchable in Azure Cognitive Search. Inputs to the indexer are your entities, in a single table. Output is a search index with searchable content and metadata stored in individual fields.

This article supplements **Create an indexer** with information that's specific to indexing from Azure Table Storage. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Prerequisites

- Azure Table Storage
- Tables containing text. If you have binary data, consider AI enrichment for image analysis.
- Read permissions on Azure Storage. A "full access" connection string includes a key that gives access to the content, but if you're using Azure roles, make sure the search service managed identity has **Data and Reader** permissions.
- Use a REST client, such as Postman app, if you want to formulate REST calls similar to the ones shown in this article.

Define the data source

The data source definition specifies the source data to index, credentials, and policies for change detection. A data source is an independent resource that can be used by multiple indexers.

1. Create or update a data source to set its definition:

HTTP

```
 POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
 {
     "name": "my-table-storage-ds",
     "description": null,
     "type": "azuretable",
```

```
      "subtype": null,
      "credentials": {
         "connectionString":
 "DefaultEndpointsProtocol=https;AccountName=<account name>"
      },
      "container": {
         "name": "my-table-in-azure-storage",
         "query": ""
      },
      "dataChangeDetectionPolicy": null,
      "dataDeletionDetectionPolicy": null,
      "encryptionKey": null,
      "identity": null
  }
```

2. Set "type" to `"azuretable"` (required).

3. Set "credentials" to an Azure Storage connection string. The next section describes the supported formats.

4. Set "container" to the name of the table.

5. Optionally, set "query" to a filter on PartitionKey. Setting this property is a best practice that improves performance. If "query" is null, the indexer executes a full table scan, which can result in poor performance if the tables are large.

A data source definition can also include soft deletion policies, if you want the indexer to delete a search document when the source document is flagged for deletion.

Supported credentials and connection strings

Indexers can connect to a table using the following connections.

Full access storage account connection string
`{ "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>;" }`
You can get the connection string from the Storage account page in Azure portal by selecting **Access keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Storage/storageAccounts/<your storage account name>/;" }`

Managed identity connection string
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity.

Storage account shared access signature** (SAS) connection string
`{ "connectionString" : "BlobEndpoint=https://<your account>.blob.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=b&sp=rl;" }`
The SAS should have the list and read permissions on tables and entities.

Container shared access signature
`{ "connectionString" : "ContainerSharedAccessUri=https://<your storage account>.blob.core.windows.net/<container name>?sv=2016-05-31&sr=c&sig=<the signature>&se=<the validity end time>&sp=rl;" }`
The SAS should have the list and read permissions on the container. For more information, see Using Shared Access Signatures.

> **Note**
>
> If you use SAS credentials, you'll need to update the data source credentials periodically with renewed signatures to prevent their expiration. When SAS credentials expire, the indexer will fail with an error message similar to "Credentials provided in the connection string are invalid or have expired".

Partition for improved performance

By default, Azure Cognitive Search uses the following internal query filter to keep track of which source entities have been updated since the last run: `Timestamp >= HighWaterMarkValue`. Because Azure tables don't have a secondary index on the `Timestamp` field, this type of query requires a full table scan and is therefore slow for large tables.

To avoid a full scan, you can use table partitions to narrow the scope of each indexer job.

- If your data can naturally be partitioned into several partition ranges, create a data source and a corresponding indexer for each partition range. Each indexer now has to process only a specific partition range, resulting in better query performance. If

the data that needs to be indexed has a small number of fixed partitions, even better: each indexer only does a partition scan.

For example, to create a data source for processing a partition range with keys from `000` to `100`, use a query like this: `"container" : { "name" : "my-table", "query" : "PartitionKey ge '000' and PartitionKey lt '100' " }`

- If your data is partitioned by time (for example, if you create a new partition every day or week), consider the following approach:
  - In the data source definition, specify a query similar to the following example: `(PartitionKey ge <TimeStamp>) and (other filters)`.
  - Monitor indexer progress by using Get Indexer Status API, and periodically update the `<TimeStamp>` condition of the query based on the latest successful high-water-mark value.
  - With this approach, if you need to trigger a full reindex, reset the data source query in addition to resetting the indexer.

Add search fields to an index

In a search index, add fields to accept the content and metadata of your table entities.

1. Create or update an index to define search fields that will store content from entities:

   HTTP

   ```
   POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
   {
     "name" : "my-search-index",
     "fields": [
       { "name": "Key", "type": "Edm.String", "key": true, "searchable": false },
       { "name": "SomeColumnInMyTable", "type": "Edm.String", "searchable": true }
     ]
   }
   ```

2. Create a document key field ("key": true), but allow the indexer to populate it automatically. A table indexer populates the key field with concatenated partition and row keys from the table. For example, if a row's PartitionKey is `1` and RowKey

is `1_123`, then the key value is `11_123`. If the partition key is null, just the row key is used.

If you're using the Import data wizard to create the index, the portal infers a "Key" field for the search index and uses an implicit field mapping to connect the source and destination fields. You don't have to add the field yourself, and you don't need to set up a field mapping.

If you're using the REST APIs and you want implicit field mappings, create and name the document key field "Key" in the search index definition as shown in the previous step (`{ "name": "Key", "type": "Edm.String", "key": true, "searchable": false }`). The indexer populates the Key field automatically, with no field mappings required.

If you don't want a field named "Key" in your search index, add an explicit field mapping in the indexer definition with the field name you want, setting the source field to "Key":

JSON

```
"fieldMappings" : [
  {
    "sourceFieldName" : "Key",
    "targetFieldName" : "MyDocumentKeyFieldName"
  }
]
```

3. Now add any other entity fields that you want in your index. For example, if an entity looks like the following example, your search index should have fields for HotelName, Description, and Category to receive those values.

PartitionKey	RowKey	Timestamp	HotelName	Description	Category
	20	2022-01-14T20:19:10.29...	Travel Resort	The Best Gaming Resort ...	Budget
	23	2022-01-14T20:20:36.88...	Days Hotel	Mix and mingle in the h...	Boutique
	3	2022-01-14T20:21:20.61...	Triple Landscape Hotel	The Hotel stands out for...	Resort and Spa
	31	2022-01-14T20:20:01.02...	Santa Fe Stay	Nestled on six beautifull...	Resort and Spa

Using the same names and compatible data types minimizes the need for field mappings. When names and types are the same, the indexer can determine the data path automatically.

Configure and run the table indexer

Once you have an index and data source, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
{
    "name" : "my-table-indexer",
    "dataSourceName" : "my-table-storage-ds",
    "targetIndexName" : "my-search-index",
    "disabled": null,
    "schedule": null,
    "parameters" : {
        "batchSize" : null,
        "maxFailedItems" : null,
        "maxFailedItemsPerBatch" : null,
        "base64EncodeKeys" : null,
        "configuration" : { }
    },
    "fieldMappings" : [ ],
    "cache": null,
    "encryptionKey": null
}
```

2. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index. The Target field is the name of the field in the search index.

```JSON
 "fieldMappings" : [
   {
     "sourceFieldName" : "Description",
     "targetFieldName" : "HotelDescription"
   }
]
```

3. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2023-02-21T00:23:24.957Z",
            "endTime":"2023-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2023-02-21T00:23:24.957Z",
                "endTime":"2023-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Next steps

Learn more about how to run the indexer, monitor status, or schedule indexer execution.

The following articles apply to indexers that pull content from Azure Storage:

- Tutorial: Index JSON blobs from Azure Storage
- Tutorial: Index encrypted blobs in Azure Storage

Change and delete detection using indexers for Azure Storage in Azure Cognitive Search

Article • 11/09/2022 • 5 minutes to read

After an initial search index is created, you might want subsequent indexer jobs to only pick up new and changed documents. For indexed content that originates from Azure Storage, change detection occurs automatically because indexers keep track of the last update using the built-in timestamps on objects and files in Azure Storage.

Although change detection is a given, deletion detection isn't. An indexer doesn't track object deletion in data sources. To avoid having orphan search documents, you can implement a "soft delete" strategy that results in deleting search documents first, with physical deletion in Azure Storage following as a second step.

There are two ways to implement a soft delete strategy:

- Native blob soft delete (preview), applies to Blob Storage only
- Soft delete using custom metadata

Prerequisites

- Use an Azure Storage indexer for Blob Storage, Table Storage, File Storage, or Data Lake Storage Gen2
- Use consistent document keys and file structure. Changing document keys or directory names and paths (applies to ADLS Gen2) breaks the internal tracking information used by indexers to know which content was indexed, and when it was last indexed.

> **Note**
>
> ADLS Gen2 allows directories to be renamed. When a directory is renamed, the timestamps for the blobs in that directory do not get updated. As a result, the indexer will not re-index those blobs. If you need the blobs in a directory to be reindexed after a directory rename because they now have new URLs, you will need to update the `LastModified` timestamp for all the blobs in the directory so that the indexer knows to re-index them during a future run. The virtual directories in Azure Blob Storage cannot be changed, so they do not have this issue.

Native blob soft delete (preview)

For this deletion detection approach, Cognitive Search depends on the native blob soft delete feature in Azure Blob Storage to determine whether blobs have transitioned to a soft deleted state. When blobs are detected in this state, a search indexer uses this information to remove the corresponding document from the index.

> **Important**
>
> Support for native blob soft delete is in preview under **Supplemental Terms of Use**. The **REST API version 2020-06-30-Preview** provides this feature. There's currently no .NET SDK support.

Requirements for native soft delete

- Blobs must be in an Azure Blob Storage container. The Cognitive Search native blob soft delete policy isn't supported for blobs in ADLS Gen2.
- Enable soft delete for blobs.
- Document keys for the documents in your index must be mapped to either be a blob property or blob metadata, such as "metadata_storage_path".
- You must use the preview REST API (`api-version=2020-06-30-Preview`), or the indexer Data Source configuration in the Azure portal, to configure support for soft delete.
- Blob versioning must not be enabled in the storage account. Otherwise, native soft delete isn't supported by design.

Configure native soft delete

In Blob storage, when enabling soft delete per the requirements, set the retention policy to a value that's much higher than your indexer interval schedule. If there's an issue running the indexer, or if you have a large number of documents to index, there's plenty of time for the indexer to eventually process the soft deleted blobs. Azure Cognitive Search indexers will only delete a document from the index if it processes the blob while it's in a soft deleted state.

In Cognitive Search, set a native blob soft deletion detection policy on the data source. You can do this either from the Azure portal or by using preview REST API (`api-`

`version=2020-06-30-Preview`). The following instructions explain how to set the delete detection policy in Azure portal or through REST APIs.

Azure portal

1. Sign in to Azure portal.

2. On the Cognitive Search service Overview page, go to **New Data Source**, a visual editor for specifying a data source definition.

   The following screenshot shows where you can find this feature in the portal.

3. On the **New Data Source** form, fill out the required fields, select the **Track deletions** checkbox and choose **Native blob soft delete**. Then hit **Save** to enable the feature on Data Source creation.

Reindex undeleted blobs using native soft delete policies

If you restore a soft deleted blob in Blob storage, the indexer won't always reindex it. This is because the indexer uses the blob's `LastModified` timestamp to determine whether indexing is needed. When a soft deleted blob is undeleted, its `LastModified` timestamp doesn't get updated, so if the indexer has already processed blobs with more recent `LastModified` timestamps, it won't reindex the undeleted blob.

To make sure that an undeleted blob is reindexed, you'll need to update the blob's `LastModified` timestamp. One way to do this is by resaving the metadata of that blob. You don't need to change the metadata, but resaving the metadata will update the blob's `LastModified` timestamp so that the indexer knows to pick it up.

Soft delete strategy using custom metadata

This method uses custom metadata to indicate whether a search document should be removed from the index. It requires two separate actions: deleting the search document from the index, followed by file deletion in Azure Storage.

There are steps to follow in both Azure Storage and Cognitive Search, but there are no other feature dependencies.

1. In Azure Storage, add a custom metadata key-value pair to the file to indicate the file is flagged for deletion. For example, you could name the property "IsDeleted", set to false. When you want to delete the file, change it to true.

2. In Azure Cognitive Search, edit the data source definition to include a "dataDeletionDetectionPolicy" property. For example, the following policy considers a file to be deleted if it has a metadata property `IsDeleted` with the value `true`:

   HTTP

   ```
   PUT https://[service name].search.windows.net/datasources/file-
   datasource?api-version=2020-06-30
   {
       "name" : "file-datasource",
       "type" : "azurefile",
       "credentials" : { "connectionString" : "<your storage connection
   string>" },
       "container" : { "name" : "my-share", "query" : null },
       "dataDeletionDetectionPolicy" : {
           "@odata.type"
   :"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
           "softDeleteColumnName" : "IsDeleted",
   ```

```
            "softDeleteMarkerValue" : "true"
        }
    }
```

3. Run the indexer. Once the indexer has processed the file and deleted the document from the search index, you can then delete the physical file in Azure Storage.

Reindex undeleted blobs and files

You can reverse a soft-delete if the original source file still physically exists in Azure Storage.

1. Change the `"softDeleteMarkerValue" : "false"` on the blob or file in Azure Storage.

2. Check the blob or file's `LastModified` timestamp to make it's newer than the last indexer run. You can force an update to the current date and time by resaving the existing metadata.

3. Run the indexer.

Next steps

- Indexers in Azure Cognitive Search
- How to configure a blob indexer
- Blob indexing overview

Import data from Azure Cosmos DB for NoSQL for queries in Azure Cognitive Search

Article • 02/28/2023 • 9 minutes to read

In this article, learn how to configure an **indexer** that imports content from Azure Cosmos DB for NoSQL and makes it searchable in Azure Cognitive Search.

This article supplements **Create an indexer** with information that's specific to Cosmos DB. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that Azure Cosmos DB indexing and Cognitive Search indexing are different operations. Indexing in Cognitive Search creates and loads a search index on your search service.

Prerequisites

- An Azure Cosmos DB account, database, container and items. Use the same region for both Cognitive Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An automatic indexing policy on the Azure Cosmos DB collection, set to Consistent. This is the default configuration. Lazy indexing isn't recommended and may result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the search service managed identity has **Cosmos DB Account Reader Role** permissions.
- A REST client, such as Postman, to send REST calls that create the data source, index, and indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create or update a data source to set its definition:

```
HTTP

POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]
{
    "name": "[my-cosmosdb-ds]",
    "type": "cosmosdb",
    "credentials": {
      "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name]"
    },
    "container": {
      "name": "[my-cosmos-db-collection]",
      "query": null
    },
    "dataChangeDetectionPolicy": {
      "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "  highWaterMarkColumnName": "_ts"
    },
    "dataDeletionDetectionPolicy": null,
    "encryptionKey": null,
    "identity": null
}
```

2. Set "type" to `"cosmosdb"` (required). If you're using an older Search API version 2017-11-11, the syntax for "type" is `"documentdb"`. Otherwise, for 2019-05-06 and later, use `"cosmosdb"`.

3. Set "credentials" to a connection string. The next section describes the supported formats.

4. Set "container" to the collection. The "name" property is required and it specifies the ID of the database collection to be indexed. The "query" property is optional. Use it to flatten an arbitrary JSON document into a flat schema that Azure Cognitive Search can index.

5. Set "dataChangeDetectionPolicy" if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.

6. Set "dataDeletionDetectionPolicy" if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections. For connections that target the SQL API, you can omit "ApiKind" from the connection string.

Avoid port numbers in the endpoint URL. If you include the port number, the connection will fail.

Full access connection string
`{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>" }`
You can get the connection string from the Azure Cosmos DB account page in the Azure portal by selecting **Keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind];)" }`
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity and created a role assignment that grants **Cosmos DB Account Reader Role** permissions. See Setting up an indexer connection to an Azure Cosmos DB database using a managed identity for more information.

Using queries to shape indexed data

In the "query" property under "container", you can specify a SQL query to flatten nested properties or arrays, project JSON properties, and filter the data to be indexed.

Example document:

HTTP

```
    {
        "userId": 10001,
        "contact": {
            "firstName": "andy",
            "lastName": "hoh"
        },
        "company": "microsoft",
```

```
        "tags": ["azure", "cosmosdb", "search"]
    }
```

Filter query:

```SQL
SELECT * FROM c WHERE c.company = "microsoft" and c._ts >= @HighWaterMark
ORDER BY c._ts
```

Flattening query:

```SQL
SELECT c.id, c.userId, c.contact.firstName, c.contact.lastName, c.company,
c._ts FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

Projection query:

```SQL
SELECT VALUE { "id":c.id, "Name":c.contact.firstName, "Company":c.company,
"_ts":c._ts } FROM c WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

Array flattening query:

```SQL
SELECT c.id, c.userId, tag, c._ts FROM c JOIN tag IN c.tags WHERE c._ts >=
@HighWaterMark ORDER BY c._ts
```

Unsupported queries (DISTINCT and GROUP BY)

Queries using the DISTINCT keyword or GROUP BY clause aren't supported. Azure Cognitive Search relies on SQL query pagination to fully enumerate the results of the query. Neither the DISTINCT keyword or GROUP BY clauses are compatible with the continuation tokens used to paginate results.

Examples of unsupported queries:

```SQL
SELECT DISTINCT c.id, c.userId, c._ts FROM c WHERE c._ts >= @HighWaterMark
ORDER BY c._ts
```

```
SELECT DISTINCT VALUE c.name FROM c ORDER BY c.name

SELECT TOP 4 COUNT(1) AS foodGroupCount, f.foodGroup FROM Food f GROUP BY
f.foodGroup
```

Although Azure Cosmos DB has a workaround to support SQL query pagination with the DISTINCT keyword by using the ORDER BY clause, it isn't compatible with Azure Cognitive Search. The query will return a single JSON value, whereas Azure Cognitive Search expects a JSON object.

SQL

```
-- The following query returns a single JSON value and isn't supported by
Azure Cognitive Search
SELECT DISTINCT VALUE c.name FROM c ORDER BY c.name
```

Add search fields to an index

In a search index, add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with source data. For content in Azure Cosmos DB, your search index schema should correspond to the Azure Cosmos DB items in your data source.

1. Create or update an index to define search fields that will store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-
version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]
{
    "name": "mysearchindex",
    "fields": [{
        "name": "rid",
        "type": "Edm.String",
        "key": true,
        "searchable": false
    },
    {
        "name": "description",
        "type": "Edm.String",
        "filterable": false,
        "searchable": true,
        "sortable": false,
        "facetable": false,
        "suggestions": true
```

```
    }
  ]
}
```

2. Create a document key field ("key": true). For partitioned collections, the default document key is the Azure Cosmos DB `_rid` property, which Azure Cognitive Search automatically renames to `rid` because field names can't start with an underscore character. Also, Azure Cosmos DB `_rid` values contain characters that are invalid in Azure Cognitive Search keys. For this reason, the `_rid` values are Base64 encoded.

3. Create additional fields for more searchable content. See Create an index for details.

Mapping data types

JSON data types	Cognitive Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-cosmosdb-indexer]",
    "dataSourceName" : "[my-cosmosdb-ds]",
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {}
        },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

3. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

```HTTP
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status":"running",
    "lastResult": {
        "status":"success",
        "errorMessage":null,
        "startTime":"2022-02-21T00:23:24.957Z",
        "endTime":"2022-02-21T00:36:47.752Z",
        "errors":[],
        "itemsProcessed":1599501,
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
    },
    "executionHistory":
    [
        {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        ... earlier history items
    ]
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the HighWaterMarkChangeDetectionPolicy using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a data source definition with a change detection policy:

```HTTP
"dataChangeDetectionPolicy": {
    "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
"  highWaterMarkColumnName": "_ts"
},
```

Incremental indexing and custom queries

If you're using a custom query to retrieve documents, make sure the query orders the results by the `_ts` column. This enables periodic check-pointing that Azure Cognitive Search uses to provide incremental progress in the presence of failures.

In some cases, even if your query contains an `ORDER BY [collection alias]._ts` clause, Azure Cognitive Search may not infer that the query is ordered by the `_ts`. You can tell Azure Cognitive Search that results are ordered by setting the `assumeOrderByHighWaterMarkColumn` configuration property.

To specify this hint, create or update your indexer definition as follows:

```HTTP
{
    ... other indexer definition properties
    "parameters" : {
        "configuration" : { "assumeOrderByHighWaterMarkColumn" : true } }
}
```

Indexing deleted documents

When rows are deleted from the collection, you normally want to delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

```HTTP
"dataDeletionDetectionPolicy"": {
    "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName" : "the property that specifies whether a document
was deleted",
    "softDeleteMarkerValue" : "the value that identifies a document as
deleted"
}
```

If you're using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The following example creates a data source with a soft-deletion policy:

```HTTP
POST https://[service name].search.windows.net/datasources?api-version=2020-
06-30
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "[my-cosmosdb-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "AccountEndpoint=https://[cosmos-account-
name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-
database-name]"
    },
    "container": { "name": "[my-cosmos-collection]" },
    "dataChangeDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "_ts"
    },
    "dataDeletionDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName": "isDeleted",
        "softDeleteMarkerValue": "true"
    }
}
```

Use .NET

For data accessed through the SQL API protocol, you can use the .NET SDK to automate with indexers. We recommend that you review the previous REST API sections to learn

concepts, workflow, and requirements. You can then refer to following .NET API reference documentation to implement a JSON indexer in managed code:

- azure.search.documents.indexes.models.searchindexerdatasourceconnection
- azure.search.documents.indexes.models.searchindexerdatasourcetype
- azure.search.documents.indexes.models.searchindex
- azure.search.documents.indexes.models.searchindexer

Next steps

You can now control how you run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure Cosmos DB:

- Set up an indexer connection to an Azure Cosmos DB database using a managed identity
- Index large data sets
- Indexer access to content protected by Azure network security features

Import data from Azure Cosmos DB for MongoDB for queries in Azure Cognitive Search

Article • 02/28/2023 • 8 minutes to read

> **Important**
>
> MongoDB API support is currently in public preview under **supplemental Terms of Use**. Currently, there is no SDK support.

In this article, learn how to configure an **indexer** that imports content from Azure Cosmos DB for MongoDB and makes it searchable in Azure Cognitive Search.

This article supplements **Create an indexer** with information that's specific to Cosmos DB. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that Azure Cosmos DB indexing and Cognitive Search indexing are different operations. Indexing in Cognitive Search creates and loads a search index on your search service.

Prerequisites

- Register for the preview to provide feedback and get help with any issues you encounter.
- An Azure Cosmos DB account, database, collection, and documents. Use the same region for both Cognitive Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An automatic indexing policy on the Azure Cosmos DB collection, set to Consistent. This is the default configuration. Lazy indexing isn't recommended and may result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the search service managed identity has **Cosmos DB Account Reader Role** permissions.

- A REST client, such as Postman, to send REST calls that create the data source, index, and indexer.

Limitations

These are the limitations of this feature:

- Custom queries aren't supported for specifying the data set.
- The column name `_ts` is a reserved word. If you need this field, consider alternative solutions for populating an index.
- The MongoDB attribute `$ref` is a reserved word. If you need this in your MongoDB collection, consider alternative solutions for populating an index.

As an alternative to this connector, if your scenario has any of those requirements, you could use the Push API/SDK or consider Azure Data Factory with an Azure Cognitive Search index as the sink.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

For this call, specify a preview REST API version (2020-06-30-Preview or 2021-04-30-Preview) to create a data source that connects via the MongoDB API.

1. Create or update a data source to set its definition:

HTTP

```
POST https://[service name].search.windows.net/datasources?api-version=2021-04-30-Preview
Content-Type: application/json
api-key: [Search service admin key]
{
  "name": "[my-cosmosdb-mongodb-ds]",
  "type": "cosmosdb",
  "credentials": {
    "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name];ApiKind=MongoDb;"
  },
  "container": {
    "name": "[cosmos-db-collection]",
```

```
    "query": null
  },
  "dataChangeDetectionPolicy": {
    "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName": "_ts"
  },
  "dataDeletionDetectionPolicy": null,
  "encryptionKey": null,
  "identity": null
}
```

2. Set "type" to `"cosmosdb"` (required).

3. Set "credentials" to a connection string. The next section describes the supported formats.

4. Set "container" to the collection. The "name" property is required and it specifies the ID of the database collection to be indexed. For Azure Cosmos DB for MongoDB, "query" isn't supported.

5. Set "dataChangeDetectionPolicy" if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.

6. Set "dataDeletionDetectionPolicy" if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections. For connections that target the MongoDB API, be sure to include "ApiKind" in the connection string.

Avoid port numbers in the endpoint URL. If you include the port number, the connection will fail.

Full access connection string
`{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>;ApiKind=MongoDb" }`
You can get the *Cosmos DB auth key* from the Azure Cosmos DB account page in the Azure portal by selecting **Connection String** in the left navigation pane. Make sure to copy **Primary Password** and replace *Cosmos DB auth key* value with it.

Managed identity connection string

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind];)" }`
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity and created a role assignment that grants **Cosmos DB Account Reader Role** permissions. See Setting up an indexer connection to an Azure Cosmos DB database using a managed identity for more information.

Add search fields to an index

In a search index, add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with source data. For content in Azure Cosmos DB, your search index schema should correspond to the Azure Cosmos DB items in your data source.

1. Create or update an index to define search fields that will store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "mysearchindex",
    "fields": [{
        "name": "doc_id",
        "type": "Edm.String",
        "key": true,
        "retrievable": true,
        "searchable": false
    }, {
        "name": "description",
        "type": "Edm.String",
        "filterable": false,
        "searchable": true,
        "sortable": false,
        "facetable": false,
        "suggestions": true
    }]
}
```

2. Create a document key field ("key": true). For a search index based on a MongoDB collection, the document key can be "doc_id", "rid", or some other string field that contains unique values. As long as field names and data types are the same on both sides, no field mappings are required.

    - "doc_id" represents "_id" for the object identifier. If you specify a field of "doc_id" in the index, the indexer populates it with the values of the object identifier.
    - "rid" is a system property in Azure Cosmos DB. If you specify a field of "rid" in the index, the indexer populates it with the base64-encoded value of the "rid" property.
    - For any other field, your search field should have the same name as defined in the collection.

3. Create additional fields for more searchable content. See Create an index for details.

Mapping data types

JSON data type	Cognitive Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

   HTTP

   ```
   POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
   Content-Type: application/json
   api-key: [search service admin key]
   {
       "name" : "[my-cosmosdb-indexer]",
       "dataSourceName" : "[my-cosmosdb-mongodb-ds]",
       "targetIndexName" : "[my-search-index]",
       "disabled": null,
       "schedule": null,
       "parameters": {
           "batchSize": null,
           "maxFailedItems": 0,
           "maxFailedItemsPerBatch": 0,
           "base64EncodeKeys": false,
           "configuration": {}
           },
       "fieldMappings": [],
       "encryptionKey": null
   }
   ```

2. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

3. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
```

```
Content-Type: application/json
api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status":"running",
    "lastResult": {
        "status":"success",
        "errorMessage":null,
        "startTime":"2022-02-21T00:23:24.957Z",
        "endTime":"2022-02-21T00:36:47.752Z",
        "errors":[],
        "itemsProcessed":1599501,
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
    },
    "executionHistory":
    [
        {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        ... earlier history items
    ]
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism

is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the HighWaterMarkChangeDetectionPolicy using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a data source definition with a change detection policy:

```HTTP
"dataChangeDetectionPolicy": {
    "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
"  highWaterMarkColumnName": "_ts"
},
```

Indexing deleted documents

When rows are deleted from the collection, you normally want to delete those rows from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

```HTTP
"dataDeletionDetectionPolicy"": {
    "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName" : "the property that specifies whether a document
was deleted",
    "softDeleteMarkerValue" : "the value that identifies a document as
deleted"
}
```

If you're using a custom query, make sure that the property referenced by `softDeleteColumnName` is projected by the query.

The following example creates a data source with a soft-deletion policy:

```HTTP
POST https://[service name].search.windows.net/datasources?api-version=2020-
06-30
Content-Type: application/json
api-key: [Search service admin key]
```

```
{
    "name": ["my-cosmosdb-mongodb-ds]",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "AccountEndpoint=https://[cosmos-account-
name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-
database-name];ApiKind=MongoDB"
    },
    "container": { "name": "[my-cosmos-collection]" },
    "dataChangeDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "_ts"
    },
    "dataDeletionDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName": "isDeleted",
        "softDeleteMarkerValue": "true"
    }
}
```

Next steps

You can now control how you run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure Cosmos DB:

- Set up an indexer connection to an Azure Cosmos DB database using a managed identity
- Index large data sets
- Indexer access to content protected by Azure network security features

---

Additional resources

🕮 Documentation

Azure Cosmos DB NoSQL indexer - Azure Cognitive Search

Set up a search indexer to index data stored in Azure Cosmos DB for full text search in Azure Cognitive Search. This article explains how index data using the NoSQL API protocol.

Change log for Azure Cosmos DB API for MongoDB

Notifies our customers of any minor/medium updates that were pushed

Data partitioning in Azure Cosmos DB for Gremlin

Learn how you can use a partitioned graph in Azure Cosmos DB. This article also describes the requirements and best practices for a partitioned graph.

Improve performance and optimize costs when upgrading to Azure Cosmos DB API for MongoDB 4.0+

Learn how upgrading your API for MongoDB account to versions 4.0+ saves you money on queries and storage.

Change streams in Azure Cosmos DB's API for MongoDB

Learn how to use change streams n Azure Cosmos DB's API for MongoDB to get the changes made to your data.

Azure Cosmos DB Serverless 1-TB container (preview)

Learn more about Serverless 1-TB container.

Limits of Azure Cosmos DB for Gremlin

Reference documentation for runtime limitations of Graph engine

Frequently asked questions on hierarchical partition keys in Azure Cosmos DB (preview)

Frequently asked questions about subpartitioning in Azure Cosmos DB

Show 5 more

Training

Learning path

Define and implement an indexing strategy for Azure Cosmos DB for NoSQL - Training

Define and implement an indexing strategy for Azure Cosmos DB for NoSQL

Certification

Microsoft Certified: Azure Cosmos DB Developer Specialty - Certifications

Azure Cosmos DB developers design and implement data models and data distribution, load data into an Azure Cosmos DB database, and optimize and maintain the solution.

Import data from Azure Cosmos DB for Apache Gremlin for queries in Azure Cognitive Search

Article • 02/28/2023 • 9 minutes to read

> **Important**
>
> The Azure Cosmos DB for Apache Gremlin indexer is currently in public preview under **Supplemental Terms of Use**. Currently, there is no SDK support.

In this article, learn how to configure an **indexer** that imports content from Azure Cosmos DB for Apache Gremlin and makes it searchable in Azure Cognitive Search.

This article supplements **Create an indexer** with information that's specific to Cosmos DB. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer. Data extraction occurs when you submit the Create Indexer request.

Because terminology can be confusing, it's worth noting that Azure Cosmos DB indexing and Cognitive Search indexing are different operations. Indexing in Cognitive Search creates and loads a search index on your search service.

Prerequisites

- Register for the preview to provide feedback and get help with any issues you encounter.
- An Azure Cosmos DB account, database, container, and items. Use the same region for both Cognitive Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.
- An automatic indexing policy on the Azure Cosmos DB collection, set to Consistent. This is the default configuration. Lazy indexing isn't recommended and may result in missing data.
- Read permissions. A "full access" connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the search service managed identity has **Cosmos DB Account Reader Role** permissions.

- A REST client, such as Postman, to send REST calls that create the data source, index, and indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

For this call, specify a preview REST API version (2020-06-30-Preview or 2021-04-30-Preview) to create a data source that connects via Azure Cosmos DB for Apache Gremlin.

1. Create or update a data source to set its definition:

   HTTP

   ```
   POST https://[service name].search.windows.net/datasources?api-version=2021-04-30-Preview
   Content-Type: application/json
   api-key: [Search service admin key]
   {
     "name": "[my-cosmosdb-gremlin-ds]",
     "type": "cosmosdb",
     "credentials": {
       "connectionString": "AccountEndpoint=https://[cosmos-account-name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-database-name];ApiKind=Gremlin;"
     },
     "container": {
       "name": "[cosmos-db-collection]",
       "query": "g.V()"
     },
     "dataChangeDetectionPolicy": {
       "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
       "highWaterMarkColumnName": "_ts"
     },
     "dataDeletionDetectionPolicy": null,
     "encryptionKey": null,
     "identity": null
   }
   }
   ```

2. Set "type" to `"cosmosdb"` (required).

3. Set "credentials" to a connection string. The next section describes the supported formats.

4. Set "container" to the collection. The "name" property is required and it specifies the ID of the graph.

   The "query" property is optional. By default the Azure Cognitive Search indexer for Azure Cosmos DB for Apache Gremlin makes every vertex in your graph a document in the index. Edges will be ignored. The query default is `g.V()`. Alternatively, you could set the query to only index the edges. To index the edges, set the query to `g.E()`.

5. Set "dataChangeDetectionPolicy" if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs. Incremental progress will be enabled by default using `_ts` as the high water mark column.

6. Set "dataDeletionDetectionPolicy" if you want to remove search documents from a search index when the source item is deleted.

Supported credentials and connection strings

Indexers can connect to a collection using the following connections. For connections that target Azure Cosmos DB for Apache Gremlin, be sure to include "ApiKind" in the connection string.

Avoid port numbers in the endpoint URL. If you include the port number, the connection will fail.

Full access connection string
`{ "connectionString" : "AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>;ApiKind=MongoDb" }`
You can get the connection string from the Azure Cosmos DB account page in Azure portal by selecting **Keys** in the left navigation pane. Make sure to select a full connection string and not just a key.

Managed identity connection string
`{ "connectionString" : "ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind];)" }`

Managed identity connection string
This connection string doesn't require an account key, but you must have previously configured a search service to connect using a managed identity and created a role assignment that grants **Cosmos DB Account Reader Role** permissions. See Setting up an indexer connection to an Azure Cosmos DB database using a managed identity for more information.

Add search fields to an index

In a search index, add fields to accept the source JSON documents or the output of your custom query projection. Ensure that the search index schema is compatible with your graph. For content in Azure Cosmos DB, your search index schema should correspond to the Azure Cosmos DB items in your data source.

1. Create or update an index to define search fields that will store data:

HTTP

```
 POST https://[service name].search.windows.net/indexes?api-
version=2020-06-30-Preview
 Content-Type: application/json
 api-key: [Search service admin key]
 {
    "name": "mysearchindex",
    "fields": [
     {
         "name": "rid",
         "type": "Edm.String",
         "facetable": false,
         "filterable": false,
         "key": true,
         "retrievable": true,
         "searchable": true,
         "sortable": false,
         "analyzer": "standard.lucene",
         "indexAnalyzer": null,
         "searchAnalyzer": null,
         "synonymMaps": [],
         "fields": []
     },{
     }, {
         "name": "label",
         "type": "Edm.String",
         "searchable": true,
         "filterable": false,
         "retrievable": true,
         "sortable": false,
         "facetable": false,
         "key": false,
         "indexAnalyzer": null,
```

```
            "searchAnalyzer": null,
            "analyzer": "standard.lucene",
            "synonymMaps": []
        }]
    }
```

2. Create a document key field ("key": true). For partitioned collections, the default document key is the Azure Cosmos DB `_rid` property, which Azure Cognitive Search automatically renames to `rid` because field names can't start with an underscore character. Also, Azure Cosmos DB `_rid` values contain characters that are invalid in Azure Cognitive Search keys. For this reason, the `_rid` values are Base64 encoded.

3. Create additional fields for more searchable content. See Create an index for details.

Mapping data types

JSON data type	Cognitive Search field types
Bool	Edm.Boolean, Edm.String
Numbers that look like integers	Edm.Int32, Edm.Int64, Edm.String
Numbers that look like floating-points	Edm.Double, Edm.String
String	Edm.String
Arrays of primitive types such as ["a", "b", "c"]	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects such as { "type": "Point", "coordinates": [long, lat] }	Edm.GeographyPoint
Other JSON objects	N/A

Configure and run the Azure Cosmos DB indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-cosmosdb-indexer]",
    "dataSourceName" : "[my-cosmosdb-gremlin-ds]",
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {}
        },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

3. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

```HTTP
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
{
    "status":"running",
    "lastResult": {
        "status":"success",
        "errorMessage":null,
        "startTime":"2022-02-21T00:23:24.957Z",
        "endTime":"2022-02-21T00:36:47.752Z",
        "errors":[],
        "itemsProcessed":1599501,
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
    },
    "executionHistory":
    [
        {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        ... earlier history items
    ]
}
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed documents

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed documents in your database.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Cosmos DB indexers, the only supported policy is the HighWaterMarkChangeDetectionPolicy using the `_ts` (timestamp) property provided by Azure Cosmos DB.

The following example shows a data source definition with a change detection policy:

```HTTP
"dataChangeDetectionPolicy": {
    "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
"  highWaterMarkColumnName": "_ts"
},
```

Indexing deleted documents

When graph data is deleted, you might want to delete its corresponding document from the search index as well. The purpose of a data deletion detection policy is to efficiently identify deleted data items and delete the full document from the index. The data deletion detection policy isn't meant to delete partial document information. Currently, the only supported policy is the `Soft Delete` policy (deletion is marked with a flag of some sort), which is specified in the data source definition as follows:

```HTTP
"dataDeletionDetectionPolicy"": {
    "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
    "softDeleteColumnName" : "the property that specifies whether a document
was deleted",
    "softDeleteMarkerValue" : "the value that identifies a document as
deleted"
}
```

The following example creates a data source with a soft-deletion policy:

```HTTP
POST https://[service name].search.windows.net/datasources?api-version=2020-
06-30-Preview
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "[my-cosmosdb-gremlin-ds]",
    "type": "cosmosdb",
```

```
    "credentials": {
        "connectionString": "AccountEndpoint=https://[cosmos-account-
name].documents.azure.com;AccountKey=[cosmos-account-key];Database=[cosmos-
database-name];ApiKind=Gremlin"
    },
    "container": { "name": "[my-cosmos-collection]" },
    "dataChangeDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "`_ts`"
    },
    "dataDeletionDetectionPolicy": {
        "@odata.type":
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName": "isDeleted",
        "softDeleteMarkerValue": "true"
    }
}
```

Even if you enable deletion detection policy, deleting complex (`Edm.ComplexType`) fields from the index is not supported. This policy requires that the 'active' column in the Gremlin database to be of type integer, string or boolean.

Mapping graph data to fields in a search index

The Azure Cosmos DB for Apache Gremlin indexer will automatically map a couple pieces of graph data:

1. The indexer will map `_rid` to an `rid` field in the index if it exists, and Base64 encode it.

2. The indexer will map `_id` to an `id` field in the index if it exists.

3. When querying your Azure Cosmos DB database using the Azure Cosmos DB for Apache Gremlin you may notice that the JSON output for each property has an `id` and a `value`. Azure Cognitive Search Azure Cosmos DB indexer will automatically map the properties `value` into a field in your search index that has the same name as the property if it exists. In the following example, 450 would be mapped to a `pages` field in the search index.

HTTP

```
    {
        "id": "Cookbook",
        "label": "book",
        "type": "vertex",
        "properties": {
```

```
            "pages": [
              {
                "id": "48cf6285-a145-42c8-a0aa-d39079277b71",
                "value": "450"
              }
            ]
        }
    }
```

You may find that you need to use Output Field Mappings in order to map your query output to the fields in your index. You'll likely want to use Output Field Mappings instead of Field Mappings since the custom query will likely have complex data.

For example, let's say that your query produces this output:

JSON

```
    [
      {
        "vertex": {
          "id": "Cookbook",
          "label": "book",
          "type": "vertex",
          "properties": {
            "pages": [
              {
                "id": "48cf6085-a211-42d8-a8ea-d38642987a71",
                "value": "450"
              }
            ],
          }
        },
        "written_by": [
          {
            "yearStarted": "2017"
          }
        ]
      }
    ]
```

If you would like to map the value of `pages` in the JSON above to a `totalpages` field in your index, you can add the following Output Field Mapping to your indexer definition:

JSON

```
    ... // rest of indexer definition
    "outputFieldMappings": [
        {
          "sourceFieldName": "/document/vertex/pages",
          "targetFieldName": "totalpages"
```

```
        }
    ]
```

Notice how the Output Field Mapping starts with `/document` and does not include a reference to the properties key in the JSON. This is because the indexer puts each document under the `/document` node when ingesting the graph data and the indexer also automatically allows you to reference the value of `pages` by simple referencing `pages` instead of having to reference the first object in the array of `pages`.

Next steps

- To learn more about Azure Cosmos DB for Apache Gremlin, see the Introduction to Azure Cosmos DB: Azure Cosmos DB for Apache Gremlin.
- For more information about Azure Cognitive Search scenarios and pricing, see the Search service page on azure.microsoft.com.
- To learn about network configuration for indexers, see the Indexer access to content protected by Azure network security features.

Index data from Azure Database for MySQL

Article • 02/23/2023 • 7 minutes to read

> **Important**
>
> MySQL support is currently in public preview under **Supplemental Terms of Use**. Use a **preview REST API** (2020-06-30-preview or later) to index your content. There is currently no portal support.

In this article, learn how to configure an **indexer** that imports content from Azure Database for MySQL and makes it searchable in Azure Cognitive Search.

This article supplements Creating indexers in Azure Cognitive Search with information that's specific to indexing files in Azure Database for MySQL. It uses the REST APIs to demonstrate a three-part workflow common to all indexers:

- Create a data source
- Create an index
- Create an indexer

When configured to include a high water mark and soft deletion, the indexer takes all changes, uploads, and deletes for your MySQL database. It reflects these changes in your search index. Data extraction occurs when you submit the Create Indexer request.

Prerequisites

- Register for the preview to provide feedback and get help with any issues you encounter.
- Azure Database for MySQL flexible server.
- A table or view that provides the content. A primary key is required. If you're using a view, it must have a high water mark column.
- Read permissions. A *full access* connection string includes a key that grants access to the content, but if you're using Azure roles, make sure the search service managed identity has **Reader** permissions on MySQL.

- A REST client, such as Postman, to send REST calls that create the data source, index, and indexer.

  You can also use the Azure SDK for .NET. You can't use the portal for indexer creation, but you can manage indexers and data sources once they're created.

For more information, see Azure Database for MySQL.

Preview limitations

Currently, change tracking and deletion detection aren't working if the date or timestamp is uniform for all rows. This limitation is a known issue to be addressed in an update to the preview. Until this issue is addressed, don't add a skillset to the MySQL indexer.

The preview doesn't support geometry types and blobs.

As noted, there's no portal support for indexer creation, but a MySQL indexer and data source can be managed in the portal once they exist. For example, you can edit the definitions, and reset, run, or schedule the indexer.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. The data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create or Update Data Source specifies the definition. Be sure to use a preview REST API version (2020-06-30-Preview or later) when creating the data source.

   HTTP

   ```
   POST https://[search service name].search.windows.net/datasources?api-version=2020-06-30-Preview
   Content-Type: application/json
   api-key: [admin key]

   {
       "name" : "hotel-mysql-ds",
       "description" : "[Description of MySQL data source]",
       "type" : "mysql",
       "credentials" : {
           "connectionString" :
               "Server=[MySQLServerName].MySQL.database.azure.com;
   Port=3306; Database=[DatabaseName]; Uid=[UserName]; Pwd=[Password];
   SslMode=Preferred;"
   ```

```
    },
    "container" : {
        "name" : "[TableName]"
    },
    "dataChangeDetectionPolicy" : {
        "@odata.type":
 "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "[HighWaterMarkColumn]"
    }
}
```

2. Set `type` to `"mysql"` (required).

3. Set `credentials` to an ADO.NET connection string. You can find connection strings in Azure portal, on the **Connection strings** page for MySQL.

4. Set `container` to the name of the table.

5. Set dataChangeDetectionPolicy if data is volatile and you want the indexer to pick up just the new and updated items on subsequent runs.

6. Set dataDeletionDetectionPolicy if you want to remove search documents from a search index when the source item is deleted.

Add search fields to an index

In a search index, add search index fields that correspond to the fields in your table.

Create or Update Index specifies the fields:

HTTP

```
{
    "name" : "hotels-mysql-ix",
    "fields": [
        { "name": "ID", "type": "Edm.String", "key": true, "searchable":
false },
        { "name": "HotelName", "type": "Edm.String", "searchable": true,
"filterable": false },
        { "name": "Category", "type": "Edm.String", "searchable": false,
"filterable": true, "sortable": true  },
        { "name": "City", "type": "Edm.String", "searchable": false,
"filterable": true, "sortable": true },
        { "name": "Description", "type": "Edm.String", "searchable": false,
"filterable": false, "sortable": false  }
    ]
```

If the primary key in the source table matches the document key (in this case, "ID"), the indexer imports the primary key as the document key.

Mapping data types

The following table maps the MySQL database to Cognitive Search equivalents. For more information, see Supported data types (Azure Cognitive Search).

> **Note**
>
> The preview does not support geometry types and blobs.

MySQL data types	Cognitive Search field types
`bool`, `boolean`	Edm.Boolean, Edm.String
`tinyint`, `smallint`, `mediumint`, `int`, `integer`, `year`	Edm.Int32, Edm.Int64, Edm.String
`bigint`	Edm.Int64, Edm.String
`float`, `double`, `real`	Edm.Double, Edm.String
`date`, `datetime`, `timestamp`	Edm.DateTimeOffset, Edm.String
`char`, `varchar`, `tinytext`, `mediumtext`, `text`, `longtext`, `enum`, `set`, `time`	Edm.String
unsigned numerical data, serial, decimal, dec, bit, blob, binary, geometry	N/A

Configure and run the MySQL indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[search service name].search.windows.net/indexers?api-version=2020-06-30
```

```
{
    "name" : "hotels-mysql-idxr",
    "dataSourceName" : "hotels-mysql-ds",
    "targetIndexName" : "hotels-mysql-ix",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": null,
        "maxFailedItemsPerBatch": null,
        "base64EncodeKeys": null,
        "configuration": { }
        },
    "fieldMappings" : [ ],
    "encryptionKey": null
}
```

2. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

An indexer runs automatically when it's created. You can prevent it from running by setting `disabled` to `true`. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
```

```
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2022-02-21T00:23:24.957Z",
                "endTime":"2022-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new and changed rows

Once an indexer has fully populated a search index, you might want subsequent indexer runs to incrementally index just the new and changed rows in your database.

To enable incremental indexing, set the `dataChangeDetectionPolicy` property in your data source definition. This property tells the indexer which change tracking mechanism is used on your data.

For Azure Database for MySQL indexers, the only supported policy is the HighWaterMarkChangeDetectionPolicy.

An indexer's change detection policy relies on having a *high water mark* column that captures the row version, or the date and time when a row was last updated. It's often a `DATE`, `DATETIME`, or `TIMESTAMP` column at a granularity sufficient for meeting the requirements of a high water mark column.

In your MySQL database, the high water mark column must meet the following requirements:

- All data inserts must specify a value for the column.

- All updates to an item also change the value of the column.
- The value of this column increases with each insert or update.
- Queries with the following `WHERE` and `ORDER BY` clauses can be executed efficiently: `WHERE [High Water Mark Column] > [Current High Water Mark Value] ORDER BY [High Water Mark Column]`

The following example shows a data source definition with a change detection policy:

HTTP

```
POST https://[search service name].search.windows.net/datasources?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
    {
        "name" : "[Data source name]",
        "type" : "mysql",
        "credentials" : { "connectionString" : "[connection string]" },
        "container" : { "name" : "[table or view name]" },
        "dataChangeDetectionPolicy" : {
            "@odata.type" :
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
            "highWaterMarkColumnName" : "[last_updated column name]"
        }
    }
```

> **Important**
>
> If you're using a view, you must set a high water mark policy in your indexer data source.
>
> If the source table does not have an index on the high water mark column, queries used by the MySQL indexer might time out. In particular, the `ORDER BY [High Water Mark Column]` clause requires an index to run efficiently when the table contains many rows.

Indexing deleted rows

When rows are deleted from the table or view, you normally want to delete those rows from the search index as well. However, if the rows are physically removed from the table, an indexer has no way to infer the presence of records that no longer exist. The solution is to use a *soft-delete* technique to logically delete rows without removing them from the table. Add a column to your table or view and mark rows as deleted using that column.

Given a column that provides deletion state, an indexer can be configured to remove any search documents for which deletion state is set to `true`. The configuration property that supports this behavior is a data deletion detection policy, which is specified in the data source definition as follows:

```HTTP
{
    …,
    "dataDeletionDetectionPolicy" : {
        "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName" : "[a column name]",
        "softDeleteMarkerValue" : "[the value that indicates that a row is
deleted]"
    }
}
```

The `softDeleteMarkerValue` must be a string. For example, if you have an integer column where deleted rows are marked with the value 1, use `"1"`. If you have a `BIT` column where deleted rows are marked with the Boolean true value, use the string literal `True` or `true` (the case doesn't matter).

Next steps

You can now run the indexer, monitor status, or schedule indexer execution. The following articles apply to indexers that pull content from Azure MySQL:

- Index large data sets
- Indexer access to content protected by Azure network security features

---

Additional resources

Documentation

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Index large data sets for full text search - Azure Cognitive Search

Strategies for large data indexing or computationally intensive indexing through batch mode, resourcing, and scheduled, parallel, and distributed indexing.

Create Data Source (Azure Cognitive Search REST API)

Data source connection information used by an Azure Cognitive Search indexer when crawling external data sources.

Create Index (Azure Cognitive Search REST API)

Define an index schema for fields and other constructs in an Azure Cognitive Search index.

Get Indexer Status (Azure Cognitive Search REST API)

Return current and historical status notifications generated by an Azure Cognitive Search indexer when it crawls an external data source for searchable content.

Drop and rebuild an index - Azure Cognitive Search

Add new elements, update existing elements or documents, or delete obsolete documents in a full rebuild or partial indexing to refresh an Azure Cognitive Search index.

Full text query and indexing engine architecture (Lucene) - Azure Cognitive Search

Explore Lucene query processing and document retrieval concepts for full text search, as related to Azure Cognitive Search.

Show 5 more

Index data from Azure SQL

Article • 01/19/2023 • 13 minutes to read

In this article, learn how to configure an **indexer** that imports content from Azure SQL Database or an Azure SQL managed instance and makes it searchable in Azure Cognitive Search.

This article supplements **Create an indexer** with information that's specific to Azure SQL. It uses the REST APIs to demonstrate a three-part workflow common to all indexers: create a data source, create an index, create an indexer.

This article also provides:

- A description of the change detection policies supported by the Azure SQL indexer so that you can set up incremental indexing.
- A frequently-asked-questions (FAQ) section for answers to questions about feature compatibility.

> **Note**
>
> Real-time data synchronization isn't possible with an indexer. An indexer can reindex your table at most every five minutes. If data updates need to be reflected in the index sooner, we recommend **pushing updated rows directly**.

Prerequisites

- An Azure SQL database with data in a single table or view.

  Use a table if your data is large or if you need incremental indexing using SQL's native change detection capabilities.

  Use a view if you need to consolidate data from multiple tables. Large views aren't ideal for SQL indexer. A workaround is to create a new table just for ingestion into your Cognitive Search index. You'll be able to use SQL integrated change tracking, which is easier to implement than High Water Mark.

- Read permissions. Azure Cognitive Search supports SQL Server authentication, where the user name and password are provided on the connection string. Alternatively, you can set up a managed identity and use Azure roles.

To work through the examples in this article, you'll need a REST client, such as Postman.

Other approaches for creating an Azure SQL indexer include Azure SDKs or Import data wizard in the Azure portal. If you're using Azure portal, make sure that access to all public networks is enabled in the Azure SQL firewall and that the client has access via an inbound rule.

Define the data source

The data source definition specifies the data to index, credentials, and policies for identifying changes in the data. A data source is defined as an independent resource so that it can be used by multiple indexers.

1. Create data source or Update data source to set its definition:

```HTTP
 POST https://myservice.search.windows.net/datasources?api-version=2020-06-30
 Content-Type: application/json
 api-key: admin-key

 {
     "name" : "myazuresqldatasource",
     "description" : "A database for testing Azure Cognitive Search indexes.",
     "type" : "azuresql",
     "credentials" : { "connectionString" : "Server=tcp:<your server>.database.windows.net,1433;Database=<your database>;User ID=<your user name>;Password=<your password>;Trusted_Connection=False;Encrypt=True;Connection Timeout=30;" },
     "container" : {
         "name" : "name of the table or view that you want to index",
         "query" : null (not supported in the Azure SQL indexer)
         },
     "dataChangeDetectionPolicy": null,
     "dataDeletionDetectionPolicy": null,
     "encryptionKey": null,
     "identity": null
 }
```

2. Provide a unique name for the data source that follows Azure Cognitive Search naming conventions.

3. Set "type" to `"azuresql"` (required).

4. Set "credentials" to a connection string:

- You can get a full access connection string from the Azure portal. Use the `ADO.NET connection string` option. Set the user name and password.
- Alternatively, you can specify a managed identity connection string that doesn't include database secrets with the following format: `Initial Catalog|Database=<your database name>;ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.Sql/servers/<your SQL Server name>/;Connection Timeout=connection timeout length;`.

For more information, see Connect to Azure SQL Database indexer using a managed identity.

Add search fields to an index

In a search index, add fields that correspond to the fields in SQL database. Ensure that the search index schema is compatible with source schema by using equivalent data types.

1. Create or update an index to define search fields that will store data:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]
{
    "name": "mysearchindex",
    "fields": [{
        "name": "id",
        "type": "Edm.String",
        "key": true,
        "searchable": false
    },
    {
        "name": "description",
        "type": "Edm.String",
        "filterable": false,
        "searchable": true,
        "sortable": false,
        "facetable": false,
        "suggestions": true
    }
  ]
}
```

2. Create a document key field ("key": true) that uniquely identifies each search document. This is the only field that's required in a search index. Typically, the table's primary key is mapped to the index key field. The document key must be unique and non-null. The values can be numeric in source data, but in a search index, a key is always a string.

3. Create more fields to add more searchable content. See Create an index for guidance.

Mapping data types

SQL data type	Cognitive Search field types	Notes
bit	Edm.Boolean, Edm.String	
int, smallint, tinyint	Edm.Int32, Edm.Int64, Edm.String	
bigint	Edm.Int64, Edm.String	
real, float	Edm.Double, Edm.String	
smallmoney, money decimal numeric	Edm.String	Azure Cognitive Search doesn't support converting decimal types into `Edm.Double` because doing so would lose precision
char, nchar, varchar, nvarchar	Edm.String Collection(Edm.String)	A SQL string can be used to populate a Collection(`Edm.String`) field if the string represents a JSON array of strings: `["red", "white", "blue"]`
smalldatetime, datetime, datetime2, date, datetimeoffset	Edm.DateTimeOffset, Edm.String	
uniqueidentifer	Edm.String	
geography	Edm.GeographyPoint	Only geography instances of type POINT with SRID 4326 (which is the default) are supported
rowversion	Not applicable	Row-version columns can't be stored in the search index, but they can be used for change tracking

SQL data type	Cognitive Search field types	Notes
time, timespan, binary, varbinary, image, xml, geometry, CLR types	Not applicable	Not supported

Configure and run the Azure SQL indexer

Once the index and data source have been created, you're ready to create the indexer. Indexer configuration specifies the inputs, parameters, and properties controlling run time behaviors.

1. Create or update an indexer by giving it a name and referencing the data source and target index:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [search service admin key]
{
    "name" : "[my-sqldb-indexer]",
    "dataSourceName" : "[my-sqldb-ds]",
    "targetIndexName" : "[my-search-index]",
    "disabled": null,
    "schedule": null,
    "parameters": {
        "batchSize": null,
        "maxFailedItems": 0,
        "maxFailedItemsPerBatch": 0,
        "base64EncodeKeys": false,
        "configuration": {
            "queryTimeout": "00:04:00",
            "convertHighWaterMarkToRowVersion": false,
            "disableOrderByHighWaterMarkColumn": false
        }
    },
    "fieldMappings": [],
    "encryptionKey": null
}
```

2. Under parameters, the configuration section has parameters that are specific to Azure SQL:

    - Default query timeout for SQL query execution is 5 minutes, which you can override.

- "convertHighWaterMarkToRowVersion" optimizes for the High Water Mark change detection policy. Change detection policies are set in the data source. If you're using the native change detection policy, this parameter has no effect.

- "disableOrderByHighWaterMarkColumn" causes the SQL query used by the high water mark policy to omit the ORDER BY clause. If you're using the native change detection policy, this parameter has no effect.

3. Specify field mappings if there are differences in field name or type, or if you need multiple versions of a source field in the search index.

4. See Create an indexer for more information about other properties.

An indexer runs automatically when it's created. You can prevent this by setting "disabled" to true. To control indexer execution, run an indexer on demand or put it on a schedule.

Check indexer status

To monitor the indexer status and execution history, send a Get Indexer Status request:

HTTP

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
  Content-Type: application/json
  api-key: [admin key]
```

The response includes status and the number of items processed. It should look similar to the following example:

JSON

```
    {
        "status":"running",
        "lastResult": {
            "status":"success",
            "errorMessage":null,
            "startTime":"2022-02-21T00:23:24.957Z",
            "endTime":"2022-02-21T00:36:47.752Z",
            "errors":[],
            "itemsProcessed":1599501,
            "itemsFailed":0,
            "initialTrackingState":null,
            "finalTrackingState":null
        },
```

```
        "executionHistory":
        [
            {
                "status":"success",
                "errorMessage":null,
                "startTime":"2022-02-21T00:23:24.957Z",
                "endTime":"2022-02-21T00:36:47.752Z",
                "errors":[],
                "itemsProcessed":1599501,
                "itemsFailed":0,
                "initialTrackingState":null,
                "finalTrackingState":null
            },
            ... earlier history items
        ]
    }
```

Execution history contains up to 50 of the most recently completed executions, which are sorted in the reverse chronological order so that the latest execution comes first.

Indexing new, changed, and deleted rows

If your SQL database supports change tracking, a search indexer can pick up just the new and updated content on subsequent indexer runs.

To enable incremental indexing, set the "dataChangeDetectionPolicy" property in your data source definition. This property tells the indexer which change tracking mechanism is used on your table or view.

For Azure SQL indexers, there are two change detection policies:

- "SqlIntegratedChangeTrackingPolicy" (applies to tables only)
- "HighWaterMarkChangeDetectionPolicy" (works for tables and views)

SQL Integrated Change Tracking Policy

We recommend using "SqlIntegratedChangeTrackingPolicy" for its efficiency and its ability to identify deleted rows.

Database requirements:

- SQL Server 2012 SP3 and later, if you're using SQL Server on Azure VMs
- Azure SQL Database or SQL Managed Instance
- Tables only (no views)
- On the database, enable change tracking for the table

- No composite primary key (a primary key containing more than one column) on the table
- No clustered indexes on the table. As a workaround, any clustered index would have to be dropped and re-created as nonclustered index, however, performance may be affected in the source compared to having a clustered index

Change detection policies are added to data source definitions. To use this policy, create or update your data source like this:

HTTP

```
POST https://myservice.search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: admin-key
    {
        "name" : "myazuresqldatasource",
        "type" : "azuresql",
        "credentials" : { "connectionString" : "connection string" },
        "container" : { "name" : "table name" },
        "dataChangeDetectionPolicy" : {
            "@odata.type" :
"#Microsoft.Azure.Search.SqlIntegratedChangeTrackingPolicy"
    }
```

When using SQL integrated change tracking policy, don't specify a separate data deletion detection policy. The SQL integrated change tracking policy has built-in support for identifying deleted rows. However, for the deleted rows to be detected automatically, the document key in your search index must be the same as the primary key in the SQL table.

> **Note**
>
> When using **TRUNCATE TABLE** to remove a large number of rows from a SQL table, the indexer needs to be **reset** to reset the change tracking state to pick up row deletions.

High Water Mark Change Detection policy

This change detection policy relies on a "high water mark" column in your table or view that captures the version or time when a row was last updated. If you're using a view, you must use a high water mark policy.

The high water mark column must meet the following requirements:

- All inserts specify a value for the column.
- All updates to an item also change the value of the column.
- The value of this column increases with each insert or update.
- Queries with the following WHERE and ORDER BY clauses can be executed efficiently: `WHERE [High Water Mark Column] > [Current High Water Mark Value]` `ORDER BY [High Water Mark Column]`

> **Note**
>
> We strongly recommend using the **rowversion** data type for the high water mark column. If any other data type is used, change tracking isn't guaranteed to capture all changes in the presence of transactions executing concurrently with an indexer query. When using **rowversion** in a configuration with read-only replicas, you must point the indexer at the primary replica. Only a primary replica can be used for data sync scenarios.

Change detection policies are added to data source definitions. To use this policy, create or update your data source like this:

HTTP

```
POST https://myservice.search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: admin-key
    {
        "name" : "myazuresqldatasource",
        "type" : "azuresql",
        "credentials" : { "connectionString" : "connection string" },
        "container" : { "name" : "table or view name" },
        "dataChangeDetectionPolicy" : {
            "@odata.type" :
"#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
            "highWaterMarkColumnName" : "[a rowversion or last_updated
column name]"
        }
    }
```

> **Note**
>
> If the source table doesn't have an index on the high water mark column, queries used by the SQL indexer may time out. In particular, the `ORDER BY [High Water Mark Column]` clause requires an index to run efficiently when the table contains many rows.

convertHighWaterMarkToRowVersion

If you're using a rowversion data type for the high water mark column, consider setting the `convertHighWaterMarkToRowVersion` property in indexer configuration. Setting this property to true results in the following behaviors:

- Uses the rowversion data type for the high water mark column in the indexer SQL query. Using the correct data type improves indexer query performance.
- Subtracts one from the rowversion value before the indexer query runs. Views with one-to-many joins may have rows with duplicate rowversion values. Subtracting one ensures the indexer query doesn't miss these rows.

To enable this property, create or update the indexer with the following configuration:

HTTP

```
    {
      ... other indexer definition properties
     "parameters" : {
            "configuration" : { "convertHighWaterMarkToRowVersion" : true }
}
    }
```

queryTimeout

If you encounter timeout errors, set the `queryTimeout` indexer configuration setting to a value higher than the default 5-minute timeout. For example, to set the timeout to 10 minutes, create or update the indexer with the following configuration:

HTTP

```
    {
      ... other indexer definition properties
     "parameters" : {
            "configuration" : { "queryTimeout" : "00:10:00" } }
    }
```

disableOrderByHighWaterMarkColumn

You can also disable the `ORDER BY [High Water Mark Column]` clause. However, this isn't recommended because if the indexer execution is interrupted by an error, the indexer has to re-process all rows if it runs later, even if the indexer has already processed

almost all the rows at the time it was interrupted. To disable the `ORDER BY` clause, use the `disableOrderByHighWaterMarkColumn` setting in the indexer definition:

```HTTP
    {
     ... other indexer definition properties
     "parameters" : {
            "configuration" : { "disableOrderByHighWaterMarkColumn" : true }
}
    }
```

Soft Delete Column Deletion Detection policy

When rows are deleted from the source table, you probably want to delete those rows from the search index as well. If you use the SQL integrated change tracking policy, this is taken care of for you. However, the high water mark change tracking policy doesn't help you with deleted rows. What to do?

If the rows are physically removed from the table, Azure Cognitive Search has no way to infer the presence of records that no longer exist. However, you can use the "soft-delete" technique to logically delete rows without removing them from the table. Add a column to your table or view and mark rows as deleted using that column.

When using the soft-delete technique, you can specify the soft delete policy as follows when creating or updating the data source:

```HTTP
    {
        …,
        "dataDeletionDetectionPolicy" : {
           "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
           "softDeleteColumnName" : "[a column name]",
           "softDeleteMarkerValue" : "[the value that indicates that a row
is deleted]"
        }
    }
```

The **softDeleteMarkerValue** must be a string in the JSON representation of your data source. Use the string representation of your actual value. For example, if you have an integer column where deleted rows are marked with the value 1, use `"1"`. If you have a BIT column where deleted rows are marked with the Boolean true value, use the string literal `"True"` or `"true"`, the case doesn't matter.

If you're setting up a soft delete policy from the Azure portal, don't add quotes around the soft delete marker value. The field contents are already understood as a string and will be translated automatically into a JSON string for you. In the examples above, simply type `1`, `True` or `true` into the portal's field.

FAQ

Q: Can I index Always Encrypted columns?

No. Always Encrypted columns aren't currently supported by Cognitive Search indexers.

Q: Can I use Azure SQL indexer with SQL databases running on IaaS VMs in Azure?

Yes. However, you need to allow your search service to connect to your database. For more information, see Configure a connection from an Azure Cognitive Search indexer to SQL Server on an Azure VM.

Q: Can I use Azure SQL indexer with SQL databases running on-premises?

Not directly. We don't recommend or support a direct connection, as doing so would require you to open your databases to Internet traffic. Customers have succeeded with this scenario using bridge technologies like Azure Data Factory. For more information, see Push data to an Azure Cognitive Search index using Azure Data Factory.

Q: Can I use a secondary replica in a failover cluster as a data source?

It depends. For full indexing of a table or view, you can use a secondary replica.

For incremental indexing, Azure Cognitive Search supports two change detection policies: SQL integrated change tracking and High Water Mark.

On read-only replicas, SQL Database doesn't support integrated change tracking. Therefore, you must use High Water Mark policy.

Our standard recommendation is to use the rowversion data type for the high water mark column. However, using rowversion relies on the `MIN_ACTIVE_ROWVERSION` function, which isn't supported on read-only replicas. Therefore, you must point the indexer to a primary replica if you're using rowversion.

If you attempt to use rowversion on a read-only replica, you'll see the following error:

"Using a rowversion column for change tracking isn't supported on secondary (read-only) availability replicas. Please update the datasource and specify a connection to the primary availability replica. Current database 'Updateability' property is 'READ_ONLY'".

Q: Can I use an alternative, non-rowversion column for high water mark change tracking?

It's not recommended. Only **rowversion** allows for reliable data synchronization. However, depending on your application logic, it may be safe if:

- You can ensure that when the indexer runs, there are no outstanding transactions on the table that's being indexed (for example, all table updates happen as a batch on a schedule, and the Azure Cognitive Search indexer schedule is set to avoid overlapping with the table update schedule).
- You periodically do a full reindex to pick up any missed rows.

Additional resources

🕮 Documentation

C# tutorial indexing Azure SQL data - Azure Cognitive Search

In this C# tutorial, connect to Azure SQL Database, extract searchable data, and load it into an Azure Cognitive Search index.

Indexer troubleshooting guidance - Azure Cognitive Search

This article provides indexer problem and resolution guidance for cases when no error messages are returned from the service search.

Quickstart: Search explorer query tool - Azure Cognitive Search

Search explorer is a query tool in the Azure portal that sends query requests to a search index in Azure Cognitive Search. Use it to learn syntax, test query expressions, or inspect a search document.

Drop and rebuild an index - Azure Cognitive Search

Add new elements, update existing elements or documents, or delete obsolete documents in a full rebuild or partial indexing to refresh an Azure Cognitive Search index.

Run or reset indexers - Azure Cognitive Search

Run indexers in full, or reset an indexer, skills, or individual documents to refresh all or part of a search index or knowledge store.

Index large data set using built-in indexers - Azure Cognitive Search

Strategies for large data indexing or computationally intensive indexing through batch mode, resourcing, and techniques for scheduled, parallel, and distributed indexing.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Show 5 more

Indexer connections to Azure SQL Managed Instance through a public endpoint

Article • 06/07/2022 • 2 minutes to read

If you are setting up an Azure SQL indexer that connects to an Azure SQL managed instance, you'll need to enable a public endpoint on the managed instance as a prerequisite. By default, an indexer connects to a managed instance over a public endpoint.

This article provides basic steps that include collecting information necessary for data source configuration. For more information and methodologies, see Configure public endpoint in Azure SQL Managed Instance.

> **Note**
>
> **Always Encrypted** columns are not currently supported by Cognitive Search indexers.

Enable a public endpoint

For a new SQL Managed Instance, create the resource with the **Enable public endpoint** option selected.

Alternatively, if the instance already exists, you can enable public endpoint on an existing SQL Managed Instance under **Security** > **Virtual network** > **Public endpoint** > **Enable**.

Verify NSG rules

Check the Network Security Group has the correct **Inbound security rules** that allow connections from Azure services.

Restrict inbound access to the endpoint

You can restrict inbound access to the public endpoint by replacing the current rule (`public_endpoint_inbound`) with the following two rules:

- Allowing inbound access from the `AzureCognitiveSearch` service tag ("SOURCE" = `AzureCognitiveSearch`, "NAME" = `cognitive_search_inbound`)
- Allowing inbound access from the IP address of the search service, which can be obtained by pinging its fully qualified domain name (eg., `<your-search-service-name>.search.windows.net`). ("SOURCE" = `IP address`, "NAME" = `search_service_inbound`)

For each rule, set "PORT" = `3342`, "PROTOCOL" = `TCP`, "DESTINATION" = `Any`, "ACTION" = `Allow`.

Get public endpoint connection string

Copy the connection string to use in the search indexer's data source connection. Be sure to copy the connection string for the **public endpoint** (port 3342, not port 1433).

Next steps

With configuration out of the way, you can now specify a SQL Managed Instance as an indexer data source.

Indexer connections to a SQL Server instance on an Azure virtual machine

Article • 10/18/2022 • 4 minutes to read

When configuring an Azure SQL indexer to extract content from a database on an Azure virtual machine, additional steps are required for secure connections.

A connection from Azure Cognitive Search to SQL Server instance on a virtual machine is a public internet connection. In order for secure connections to succeed, you'll need to satisfy the following requirements:

- Obtain a certificate from a Certificate Authority provider for the fully qualified domain name of the SQL Server instance on the virtual machine.
- Install the certificate on the virtual machine.

After you've installed the certificate on your VM, you're ready to complete the following steps in this article.

> **Note**
>
> **Always Encrypted** columns are not currently supported by Cognitive Search indexers.

Enable encrypted connections

Azure Cognitive Search requires an encrypted channel for all indexer requests over a public internet connection. This section lists the steps to make this work.

1. Check the properties of the certificate to verify the subject name is the fully qualified domain name (FQDN) of the Azure VM.

   You can use a tool like CertUtils or the Certificates snap-in to view the properties. You can get the FQDN from the VM service blade's Essentials section, in the **Public IP address/DNS name label** field, in the Azure portal.

   The FQDN is typically formatted as `<your-VM-name>.<region>.cloudapp.azure.com`

2. Configure SQL Server to use the certificate using the Registry Editor (regedit).

Although SQL Server Configuration Manager is often used for this task, you can't use it for this scenario. It won't find the imported certificate because the FQDN of the VM on Azure doesn't match the FQDN as determined by the VM (it identifies the domain as either the local computer or the network domain to which it's joined). When names don't match, use regedit to specify the certificate.

a. In regedit, browse to this registry key: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft SQL Server\[MSSQL13.MSSQLSERVER]\MSSQLServer\SuperSocketNetLib\Certificate`.

   The `[MSSQL13.MSSQLSERVER]` part varies based on version and instance name.

b. Set the value of the **Certificate** key to the **thumbprint** (without spaces) of the TLS/SSL certificate you imported to the VM.

There are several ways to get the thumbprint, some better than others. If you copy it from the **Certificates** snap-in in MMC, you'll probably pick up an invisible leading character as described in this support article, which results in an error when you attempt a connection. Several workarounds exist for correcting this problem. The easiest is to backspace over and then retype the first character of the thumbprint to remove the leading character in the key value field in regedit. Alternatively, you can use a different tool to copy the thumbprint.

3. Grant permissions to the service account.

   Make sure the SQL Server service account is granted appropriate permission on the private key of the TLS/SSL certificate. If you overlook this step, SQL Server won't start. You can use the **Certificates** snap-in or **CertUtils** for this task.

4. Restart the SQL Server service.

Connect to SQL Server

After you set up the encrypted connection required by Azure Cognitive Search, you'll connect to the instance through its public endpoint. The following article explains the connection requirements and syntax:

- Connect to SQL Server over the internet

Configure the network security group

It isn't unusual to configure the network security group and corresponding Azure endpoint or Access Control List (ACL) to make your Azure VM accessible to other

parties. Chances are you've done this before to allow your own application logic to connect to your SQL Azure VM. It's no different for an Azure Cognitive Search connection to your SQL Azure VM.

The links below provide instructions on NSG configuration for VM deployments. Use these instructions to ACL an Azure Cognitive Search endpoint based on its IP address.

1. Obtain the IP address of your search service. See the following section for instructions.

2. Add the search IP address to the IP filter list of the security group. Either one of following articles explains the steps:

- Tutorial: Filter network traffic with a network security group using the Azure portal

- Create, change, or delete a network security group

IP addressing can pose a few challenges that are easily overcome if you're aware of the issue and potential workarounds. The remaining sections provide recommendations for handling issues related to IP addresses in the ACL.

Restrict access to the Azure Cognitive Search

We strongly recommend that you restrict the access to the IP address of your search service and the IP address range of `AzureCognitiveSearch` service tag in the ACL instead of making your SQL Azure VMs open to all connection requests.

You can find out the IP address by pinging the FQDN (for example, `<your-search-service-name>.search.windows.net`) of your search service. Although it's possible for the search service IP address to change, it's unlikely that it will change. The IP address tends to be static for the lifetime of the service.

You can find out the IP address range of `AzureCognitiveSearch` service tag by either using Downloadable JSON files or via the Service Tag Discovery API. The IP address range is updated weekly.

Include the Azure Cognitive Search portal IP addresses

If you're using the Azure portal to create an indexer, you must grant the portal inbound access to your SQL Azure virtual machine. An inbound rule in the firewall requires that you provide the IP address of the portal.

To get the portal IP address, ping `stamp2.ext.search.windows.net`, which is the domain of the traffic manager. The request will time out, but the IP address will be visible in the status message. For example, in the message "Pinging azsyrie.northcentralus.cloudapp.azure.com [52.252.175.48]", the IP address is "52.252.175.48".

Clusters in different regions connect to different traffic managers. Regardless of the domain name, the IP address returned from the ping is the correct one to use when defining an inbound firewall rule for the Azure portal in your region.

Next steps

With configuration out of the way, you can now specify a SQL Server on Azure VM as the data source for an Azure Cognitive Search indexer. For more information, see Index data from Azure SQL.

Index data from SharePoint document libraries

Article • 04/05/2023 • 16 minutes to read

> **Important**
>
> SharePoint indexer support is currently in public preview under **Supplemental Terms of Use**. **Request access** to this feature, and after access is enabled, use a **preview REST API (2020-06-30-preview or later)** to index your content. There is currently limited portal support and no .NET SDK support.

This article explains how to configure a search indexer to index documents stored in SharePoint document libraries for full text search in Azure Cognitive Search. Configuration steps are followed by a deeper exploration of behaviors and scenarios you're likely to encounter.

Functionality

An indexer in Azure Cognitive Search is a crawler that extracts searchable data and metadata from a data source. The SharePoint indexer will connect to your SharePoint site and index documents from one or more document libraries. The indexer provides the following functionality:

- Index content and metadata from one or more document libraries.
- Incremental indexing, where the indexer identifies which file content or metadata have changed and indexes only the updated data. For example, if five PDFs are originally indexed and one is updated, only the updated PDF is indexed.
- Deletion detection is built in. If a document is deleted from a document library, the indexer will detect the delete on the next indexer run and remove the document from the index.
- Text and normalized images will be extracted by default from the documents that are indexed. Optionally a skillset can be added to the pipeline for AI enrichment.

Prerequisites

- SharePoint in Microsoft 365 cloud service
- Files in a document library

Supported document formats

The SharePoint indexer can extract text from the following document formats:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML
- ZIP

Configure the SharePoint indexer

To set up the SharePoint indexer, you'll need to perform some tasks in the Azure portal and others through the preview REST API.

The following video shows you how to set up the SharePoint indexer.
https://www.youtube-nocookie.com/embed/QmG65Vgl0JI

Step 1 (Optional): Enable system assigned managed identity

When a system-assigned managed identity is enabled, Azure creates an identity for your search service that can be used by the indexer. This identity is used to automatically detect the tenant the search service is provisioned in.

If the SharePoint site is in the same tenant as the search service, you'll need to enable the system-assigned managed identity for the search service in the Azure portal. If the SharePoint site is in a different tenant from the search service, skip this step.

After selecting **Save** you'll see an Object ID that has been assigned to your search service.

Step 2: Decide which permissions the indexer requires

The SharePoint indexer supports both delegated and application permissions. Choose which permissions you want to use based on your scenario:

- Delegated permissions, where the indexer runs under the identity of the user or app sending the request. Data access is limited to the sites and files to which the user has access. To support delegated permissions, the indexer requires a device code prompt to sign in on behalf of the user.
- Application permissions, where the indexer runs under the identity of the SharePoint tenant with access to all sites and files within the SharePoint tenant. The indexer requires a client secret to access the SharePoint tenant. The indexer will also require tenant admin approval before it can index any content.

If your Azure Active Directory organization has Conditional Access enabled and your administrator isn't able to grant any device access for Delegated permissions, you should consider Application permissions instead. For more information, see Azure Active Directory Conditional Access policies.

Step 3: Create an Azure AD application

The SharePoint indexer will use this Azure Active Directory (Azure AD) application for authentication.

1. Sign in to Azure portal.
2. Search for or navigate to **Azure Active Directory**, then select **App registrations**.

3. Select + **New registration**:
   a. Provide a name for your app.
   b. Select **Single tenant**.
   c. Skip the URI designation step. No redirect URI required.
   d. Select **Register**.

4. On the left, select **API permissions**, then **Add a permission**, then **Microsoft Graph**.

   - If the indexer is using delegated API permissions, select **Delegated permissions** and add the following:
     - **Delegated - Files.Read.All**
     - **Delegated - Sites.Read.All**
     - **Delegated - User.Read**

   Delegated permissions allow the search client to connect to SharePoint under the security identity of the current user.

   - If the indexer is using application API permissions, then select **Application permissions** and add the following:
     - **Application - Files.Read.All**
     - **Application - Sites.Read.All**

   Using application permissions means that the indexer will access the SharePoint site in a service context. So when you run the indexer it will have access to all content in the SharePoint tenant, which requires tenant admin approval. A client secret is also required for authentication. Setting up the client secret is described later in this article.

5. Give admin consent.

   Tenant admin consent is required when using application API permissions. Some tenants are locked down in such a way that tenant admin consent is required for delegated API permissions as well. If either of these conditions apply, you'll need to have a tenant admin grant consent for this Azure AD application before creating the indexer.

6. Select the **Authentication** tab.

7. Set **Allow public client flows** to **Yes** then select **Save**.

8. Select + **Add a platform**, then **Mobile and desktop applications**, then check `https://login.microsoftonline.com/common/oauth2/nativeclient`, then **Configure**.

9. (Application API Permissions only) To authenticate to the Azure AD application using application permissions, the indexer requires a client secret.

   - Select **Certificates & Secrets** from the menu on the left, then **Client secrets**, then **New client secret**.

   - In the menu that pops up, enter a description for the new client secret. Adjust the expiration date if necessary. If the secret expires, it will need to be recreated and the indexer needs to be updated with the new secret.

- The new client secret will appear in the secret list. Once you navigate away from the page the secret will no longer be visible, so copy it using the copy button and save it in a secure location.

Step 4: Create data source

> **Important**
>
> Starting in this section you need to use the preview REST API for the remaining steps. If you're not familiar with the Azure Cognitive Search REST API, we suggest taking a look at this **Quickstart**.

A data source specifies which data to index, credentials needed to access the data, and policies to efficiently identify changes in the data (new, modified, or deleted rows). A data source can be used by multiple indexers in the same search service.

For SharePoint indexing, the data source must have the following required properties:

- **name** is the unique name of the data source within your search service.
- **type** must be "sharepoint". This value is case-sensitive.
- **credentials** provide the SharePoint endpoint and the Azure AD application (client) ID. An example SharePoint endpoint is `https://microsoft.sharepoint.com/teams/MySharePointSite`. You can get the endpoint by navigating to the home page of your SharePoint site and copying the URL from the browser.
- **container** specifies which document library to index. More information on creating the container can be found in the Controlling which documents are indexed section of this document.

To create a data source, call Create Data Source using preview API version `2020-06-30-Preview` or later.

```
HTTP
```

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-datasource",
    "type" : "sharepoint",
    "credentials" : { "connectionString" : "[connection-string]" },
    "container" : { "name" : "defaultSiteLibrary", "query" : null }
}
```

Connection string format

The format of the connection string changes based on whether the indexer is using delegated API permissions or application API permissions

- Delegated API permissions connection string format

  `SharePointOnlineEndpoint=[SharePoint site url];ApplicationId=[Azure AD App ID];TenantId=[SharePoint site tenant id]`

- Application API permissions connection string format

  `SharePointOnlineEndpoint=[SharePoint site url];ApplicationId=[Azure AD App ID];ApplicationSecret=[Azure AD App client secret];TenantId=[SharePoint site tenant id]`

> **Note**
>
> If the SharePoint site is in the same tenant as the search service and system-assigned managed identity is enabled, `TenantId` doesn't have to be included in the connection string. If the SharePoint site is in a different tenant from the search service, `TenantId` must be included.

Step 5: Create an index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

To create an index, call Create Index:

HTTP

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-index",
    "fields": [
        { "name": "id", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "metadata_spo_item_name", "type": "Edm.String", "key": false, "searchable": true, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "metadata_spo_item_path", "type": "Edm.String", "key": false, "searchable": false, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "metadata_spo_item_content_type", "type": "Edm.String", "key": false, "searchable": false, "filterable":
true, "sortable": false, "facetable": true },
        { "name": "metadata_spo_item_last_modified", "type": "Edm.DateTimeOffset", "key": false, "searchable": false,
"filterable": false, "sortable": true, "facetable": false },
        { "name": "metadata_spo_item_size", "type": "Edm.Int64", "key": false, "searchable": false, "filterable": false,
"sortable": false, "facetable": false },
        { "name": "content", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable":
false }
    ]
}
```

> **Important**
>
> Only **metadata_spo_site_library_item_id** may be used as the key field in an index populated by the SharePoint indexer. If a key field doesn't exist in the data source, `metadata_spo_site_library_item_id` is automatically mapped to the key field.

Step 6: Create an indexer

An indexer connects a data source with a target search index and provides a schedule to automate the data refresh. Once the index and data source have been created, you're ready to create the indexer.

During this section you'll be asked to sign in with your organization credentials that have access to the SharePoint site. If possible, we recommend creating a new organizational user account and giving that new user the exact permissions that you want the indexer to have.

There are a few steps to creating the indexer:

1. Send a Create Indexer request:

```HTTP
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-indexer",
    "dataSourceName" : "sharepoint-datasource",
    "targetIndexName" : "sharepoint-index",
    "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null,
    "base64EncodeKeys": null,
    "configuration": {
        "indexedFileNameExtensions" : ".pdf, .docx",
        "excludedFileNameExtensions" : ".png, .jpg",
        "dataToExtract": "contentAndMetadata"
      }
    },
    "schedule" : { },
    "fieldMappings" : [
        {
          "sourceFieldName" : "metadata_spo_site_library_item_id",
          "targetFieldName" : "id",
          "mappingFunction" : {
            "name" : "base64Encode"
          }
         }
    ]
}
```

2. When creating the indexer for the first time it will fail and you'll see the following error. Go to the link in the error message. If you don't go to the link within 10 minutes the code will expire and you'll need to recreate the data source.

```HTTP
{
    "error": {
        "code": "",
        "message": "Error with data source: To sign in, use a web browser to open the page
https://microsoft.com/devicelogin and enter the code <CODE> to authenticate.  Please adjust your data source
definition in order to proceed."
    }
}
```

3. Provide the code that was included in the error message.

4. The SharePoint indexer will access the SharePoint content as the signed-in user. The user that logs in during this step will be that signed-in user. So, if you sign in with a user account that doesn't have access to a document in the Document Library that you want to index, the indexer won't have access to that document.

   If possible, we recommend creating a new user account and giving that new user the exact permissions that you want the indexer to have.

5. Approve the permissions that are being requested.

6. Resend the indexer create request. This time the request should succeed.

```http
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]

{
    "name" : "sharepoint-indexer",
    "dataSourceName" : "sharepoint-datasource",
    "targetIndexName" : "sharepoint-index",
    "parameters": {
    "batchSize": null,
    "maxFailedItems": null,
    "maxFailedItemsPerBatch": null,
```

```
    "base64EncodeKeys": null,
    "configuration:" {
        "dataToExtract": "contentAndMetadata",
        "indexedFileNameExtensions" : ".pdf, .docx",
        "excludedFileNameExtensions" : ".png, .jpg"
      }
    },
    "schedule" : { },
    "fieldMappings" : [
        {
          "sourceFieldName" : "metadata_spo_site_library_item_id",
          "targetFieldName" : "id",
          "mappingFunction" : {
            "name" : "base64Encode"
          }
        }
    ]
}
```

Note

If the Azure AD application requires admin approval and was not approved before logging in, you may see the following screen.

Admin approval is required to continue.

Step 7: Check the indexer status

After the indexer has been created, you can call Get Indexer Status:

HTTP

```
GET https://[service name].search.windows.net/indexers/sharepoint-indexer/status?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
```

Updating the data source

If there are no updates to the data source object, the indexer can run on a schedule without any user interaction. However, every time the Azure Cognitive Search data source object is updated, you'll need to sign in again in order for the indexer to run. For example, if you change the data source query, sign in again using the `https://microsoft.com/devicelogin` and a new code.

Once the data source has been updated, follow the below steps:

1. Call Run Indexer to manually kick off indexer execution.

```HTTP
POST https://[service name].search.windows.net/indexers/sharepoint-indexer/run?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
```

2. Check the indexer status. If the last indexer run has an error telling you to go to `https://microsoft.com/devicelogin`, go to that page and provide the new code.

```HTTP
GET https://[service name].search.windows.net/indexers/sharepoint-indexer/status?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
```

3. Login.

4. Manually run the indexer again and check the indexer status. This time the indexer run should successfully start.

Indexing document metadata

If you have set the indexer to index document metadata (`"dataToExtract": "contentAndMetadata"`), the following metadata will be available to index.

Identifier	Type	Description
metadata_spo_site_library_item_id	Edm.String	The combination key of site ID, library ID, and item ID which uniquely identifies an item in a document library for a site.
metadata_spo_site_id	Edm.String	The ID of the SharePoint site.
metadata_spo_library_id	Edm.String	The ID of document library.
metadata_spo_item_id	Edm.String	The ID of the (document) item in the library.
metadata_spo_item_last_modified	Edm.DateTimeOffset	The last modified date/time (UTC) of the item.
metadata_spo_item_name	Edm.String	The name of the item.
metadata_spo_item_size	Edm.Int64	The size (in bytes) of the item.
metadata_spo_item_content_type	Edm.String	The content type of the item.
metadata_spo_item_extension	Edm.String	The extension of the item.
metadata_spo_item_weburi	Edm.String	The URI of the item.
metadata_spo_item_path	Edm.String	The combination of the parent path and item name.

The SharePoint indexer also supports metadata specific to each document type. More information can be found in Content metadata properties used in Azure Cognitive Search.

> **Note**
>
> To index custom metadata, "additionalColumns" must be specified in the **query parameter of the data source**.

Include or exclude by file type

You can control which files are indexed by setting inclusion and exclusion criteria in the "parameters" section of the indexer definition.

Include specific file extensions by setting `"indexedFileNameExtensions"` to a comma-separated list of file extensions (with a leading dot). Exclude specific file extensions by setting `"excludedFileNameExtensions"` to the extensions that should be skipped. If the same extension is in both lists, it will be excluded from indexing.

```HTTP
```

```
PUT /indexers/[indexer name]?api-version=2020-06-30
{
    "parameters" : {
        "configuration" : {
            "indexedFileNameExtensions" : ".pdf, .docx",
            "excludedFileNameExtensions" : ".png, .jpeg"
        }
    }
}
```

Controlling which documents are indexed

A single SharePoint indexer can index content from one or more document libraries. Use the "container" parameter on the data source definition to indicate which sites and document libraries to index from. T The data source "container" section has two properties for this task: "name" and "query".

Name

The "name" property is required and must be one of three values:

Value	Description
defaultSiteLibrary	Index all content from the site's default document library.
allSiteLibraries	Index all content from all document libraries in a site. Document libraries from a subsite are out of scope/ If you need content from subsites, choose "useQuery" and specify "includeLibrariesInSite".
useQuery	Only index the content defined in the "query".

Query

The "query" parameter of the data source is made up of keyword/value pairs. The below are the keywords that can be used. The values are either site URLs or document library URLs.

> **Note**
>
> To get the value for a particular keyword, we recommend navigating to the document library that you're trying to include/exclude and copying the URI from the browser. This is the easiest way to get the value to use with a keyword in the query.

Keyword	Value description and examples
null	If null or empty, index either the default document library or all document libraries depending on the container name. Example: `"container" : { "name" : "defaultSiteLibrary", "query" : null }`
includeLibrariesInSite	Index content from all libraries under the specified site in the connection string. The scope includes any subsites of your site. The value should subsite. Example: `"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mycompany.sharepoint.com/mysite" }`

includeLibrary	Index all content from this library. The value is the fully qualified path to the library, which can be copied from your browser:
	
	Example 1 (fully qualified path):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/mysite/MyDocumentLibrary" }`
	
	Example 2 (URI copied from your browser):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/A`

Keyword	Value description and examples

excludeLibrary	Don't index content from this library. The value is the fully qualified path to the library, which can be copied from your browser:
	
	Example 1 (fully qualified path):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mysite.sharepoint.com/subsite1; excludeLibrary=https://mysite.sharepoint.com/subsite1/MyDocumentLibrary" }`
	
	Example 2 (URI copied from your browser):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrariesInSite=https://mycompany.sharepoint.com/teams/mysite; excludeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/AllItems.aspx" }`

additionalColumns	Index columns from the document library. The value is a comma-separated list of column names you want to index. Use a double backslash to commas in column names:
	
	Example 1 (additionalColumns=MyCustomColumn,MyCustomColumn2):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/mysite/MyDocumentLibrary;additionalColumns=MyCustomColumn,MyCustomColumn2" }`
	
	Example 2 (escape characters using double backslash):
	
	`"container" : { "name" : "useQuery", "query" : "includeLibrary=https://mycompany.sharepoint.com/teams/mysite/MyDocumentLibrary/Forms/AllItems.aspx;additionalColumns=MyCustomColumnWi }`

Handling errors

By default, the SharePoint indexer stops as soon as it encounters a document with an unsupported content type (for example, an image). You can use the `excludedFileNameExtensions` parameter to skip certain content types. However, you may need to index documents without knowing all the possible content types in advance. To continue indexing when an unsupported content type is encountered, set the `failOnUnsupportedContentType` configuration parameter to false:

HTTP

```
PUT https://[service name].search.windows.net/indexers/[indexer name]?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]

{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "failOnUnsupportedContentType" : false } }
}
```

For some documents, Azure Cognitive Search is unable to determine the content type, or unable to process a document of otherwise supported content type. To ignore this failure mode, set the `failOnUnprocessableDocument` configuration parameter to false:

HTTP

```
"parameters" : { "configuration" : { "failOnUnprocessableDocument" : false } }
```

Azure Cognitive Search limits the size of documents that are indexed. These limits are documented in Service Limits in Azure Cognitive Search. Oversized documents are treated as errors by default. However, you can still index storage metadata of oversized documents if you set `indexStorageMetadataOnlyForOversizedDocuments` configuration parameter to true:

HTTP

```
"parameters" : { "configuration" : { "indexStorageMetadataOnlyForOversizedDocuments" : true } }
```

You can also continue indexing if errors happen at any point of processing, either while parsing documents or while adding documents to an index. To ignore a specific number of errors, set the `maxFailedItems` and `maxFailedItemsPerBatch` configuration parameters to the desired values. For example:

HTTP

```
{
    ... other parts of indexer definition
    "parameters" : { "maxFailedItems" : 10, "maxFailedItemsPerBatch" : 10 }
}
```

Limitations and considerations

These are the limitations of this feature:

- Indexing SharePoint Lists is not supported.
- If a SharePoint file content and/or metadata has been indexed, renaming a SharePoint folder in its parent hierarchy is not a condition that will re-index the document.
- Indexing SharePoint .ASPX site content is not supported.
- Private endpoint is not supported.
- SharePoint supports a granular authorization model that determines per-user access at the document level. The SharePoint indexer does not pull these permissions into the search index, and Cognitive Search does not support document-level authorization. When a document is indexed from SharePoint into a search service, the content is available to anyone who has read access to the index. If you require document-level permissions, you should investigate security filters to trim results of unauthorized content.

These are the considerations when using this feature:

- If there is a requirement to implement a SharePoint content indexing solution with Cognitive Search in a production environment, consider create a custom connector using Microsoft Graph Data Connect with Blob indexer and Microsoft Graph API for incremental indexing.
- There could be Microsoft 365 processes that update SharePoint file system-metadata (based on different configurations in SharePoint) and will cause the SharePoint indexer to trigger. Make sure that you test your setup and understand the document processing count prior to using any AI enrichment. Since this is a third-party connector to Azure (since SharePoint is located in Microsoft 365), SharePoint configuration is not checked by the indexer.

See also

- Indexers in Azure Cognitive Search
- Content metadata properties used in Azure Cognitive Search

Attach a Cognitive Services resource to a skillset in Azure Cognitive Search

Article • 09/16/2022 • 5 minutes to read

When configuring an optional AI enrichment pipeline in Azure Cognitive Search, you can enrich a limited number of documents free of charge. For larger and more frequent workloads, you should attach a billable **multi-service Cognitive Services resource**.

A multi-service resource references "Cognitive Services" as the offering, rather than individual services, with access granted through a single API key. This key is specified in a **skillset** and allows Microsoft to charge you for using these APIs:

- Computer Vision for image analysis and optical character recognition (OCR)
- Language service for language detection, entity recognition, sentiment analysis, and key phrase extraction
- Translator for machine text translation

> **Tip**
>
> Azure provides infrastructure for you to monitor billing and budgets. For more information about monitoring Cognitive Services, see **Plan and manage costs for Azure Cognitive Services**.

Set the resource key

You can use the Azure portal, REST API, or an Azure SDK to attach a billable resource to a skillset.

If you leave the property unspecified, your search service will attempt to use the free enrichments available to your indexer on a daily basis. Execution of billable skills will stop at 20 transactions per indexer invocation and a "Time Out" message will appear in indexer execution history.

Azure portal

1. Sign in to Azure portal.
2. Create a multi-service Cognitive Services resource in the same region as your search service.

3. Add the key to a skillset definition:

   - If using the Import data wizard, enter the key in the second step, "Add AI enrichments".
   - If adding the key to a new or existing skillset, provide the key in the **Cognitive Services** tab.

How the key is used

Key-based billing applies when API calls to Cognitive Services resources exceed 20 API calls per indexer, per day.

The key is used for billing, but not connections. For connections, a search service connects over the internal network to a Cognitive Services resource that's co-located in the same physical region. Most regions that offer Cognitive Search also offer Cognitive Services.

If you attempt AI enrichment in a region that doesn't have both services, you'll see this message: "Provided key isn't a valid CognitiveServices type key for the region of your search service."

> **Note**
>
> Some built-in skills are based on non-regional Cognitive Services (for example, the **Text Translation Skill**). Using a non-regional skill means that your request might be serviced in a region other than the Azure Cognitive Search region. For more information on non-regional services, see the **Cognitive Services product by region** page.

Key requirements special cases

Custom Entity Lookup is metered by Azure Cognitive Search, not Cognitive Services, but it requires a Cognitive Services resource key to unlock transactions beyond 20 per indexer, per day. For this skill only, the resource key unblocks the number of transactions, but is unrelated to billing.

Free enrichments

AI enrichment offers a small quantity of free processing of billable enrichments so that you can complete short exercises without having to attach a Cognitive Services resource. Free enrichments are 20 documents per day, per indexer. You can reset the indexer to reset the counter if you want to repeat an exercise.

Some enrichments are always free:

- Utility skills that don't call Cognitive Services (namely, Conditional, Document Extraction, Shaper, Text Merge, and Text Split skills) aren't billable.
- Text extraction from PDF documents and other application files is non-billable. Text extraction occurs during the document cracking phase and isn't technically an enrichment, but it occurs during AI enrichment and is thus noted here.

Billable enrichments

During AI enrichment, Cognitive Search calls the Cognitive Services APIs for built-in skills that are based on Computer Vision, Translator, and Azure Cognitive Services for Language.

Billable built-in skills that make backend calls to Cognitive Services include Entity Linking, Entity Recognition, Image Analysis, Key Phrase Extraction, Language Detection, OCR, Personally Identifiable Information (PII) Detection, Sentiment, and Text Translation.

Image extraction is an Azure Cognitive Search operation that occurs when documents are cracked prior to enrichment. Image extraction is billable on all tiers, except for 20 free daily extractions on the free tier. Image extraction costs apply to image files inside blobs, embedded images in other files (PDF and other app files), and for images extracted using Document Extraction. For image extraction pricing, see the Azure Cognitive Search pricing page.

> **Tip**
>
> To lower the cost of skillset processing, enable **incremental enrichment (preview)** to cache and reuse any enrichments that are unaffected by changes made to a skillset. Caching requires Azure Storage (see **pricing** but the cumulative cost of skillset execution is lower if existing enrichments can be reused, especially for skillsets that use image extraction and analysis.

Example: Estimate costs

To estimate the costs associated with Cognitive Search indexing, start with an idea of what an average document looks like so you can run some numbers. For example, you might approximate:

- 1,000 PDFs.
- Six pages each.
- One image per page (6,000 images).
- 3,000 characters per page.

Assume a pipeline that consists of document cracking of each PDF, image and text extraction, optical character recognition (OCR) of images, and entity recognition of organizations.

The prices shown in this article are hypothetical. They're used to illustrate the estimation process. Your costs could be lower. For the actual prices of transactions, see See Cognitive Services pricing.

1. For document cracking with text and image content, text extraction is currently free. For 6,000 images, assume $1 for every 1,000 images extracted. That's a cost of $6.00 for this step.

2. For OCR of 6,000 images in English, the OCR cognitive skill uses the best algorithm (DescribeText). Assuming a cost of $2.50 per 1,000 images to be analyzed, you would pay $15.00 for this step.

3. For entity extraction, you'd have a total of three text records per page. Each record is 1,000 characters. Three text records per page multiplied by 6,000 pages equal 18,000 text records. Assuming $2.00 per 1,000 text records, this step would cost $36.00.

Putting it all together, you'd pay about $57.00 to ingest 1,000 PDF documents of this type with the described skillset.

Next steps

- Azure Cognitive Search pricing page
- How to define a skillset
- Create Skillset (REST)
- How to map enriched fields

Create a skillset in Azure Cognitive Search

Article • 07/14/2022 • 10 minutes to read

A skillset defines the operations that extract and enrich data to make it searchable. It executes after text and images are extracted, and after field mappings are processed.

This article explains how to create a skillset with the Create Skillset (REST API), but the same concepts and steps apply to other programming languages.

Rules for skillset definition include:

- A skillset must have a unique name within the skillset collection. When you define a skillset, you're creating a top-level resource that can be used by any indexer.
- A skillset must contain at least one skill. A typical skillset has three to five. The maximum is 30.
- A skillset can repeat skills of the same type (for example, multiple Shaper skills).
- A skillset supports chained operations, looping, and branching.

Indexers drive skillset execution. You'll need an indexer, data source, and index before you can test your skillset.

> **Tip**
>
> Enable **enrichment caching** to reuse the content you've already processed and lower the cost of development.

Add a skillset definition

Start with the basic structure. In the Create Skillset REST API, the body of the request is authored in JSON and has the following sections:

JSON

```
{
   "name":"skillset-template",
   "description":"A description makes the skillset self-documenting
```

```
(comments aren't allowed in JSON itself)",
   "skills":[

   ],
   "cognitiveServices":{
      "@odata.type":"#Microsoft.Azure.Search.CognitiveServicesByKey",
      "description":"A Cognitive Services resource in the same region as
Azure Cognitive Search",
      "key":"<Your-Cognitive-Services-Multiservice-Key>"
   },
   "knowledgeStore":{
      "storageConnectionString":"<Your-Azure-Storage-Connection-String>",
      "projections":[
         {
            "tables":[ ],
            "objects":[ ],
            "files":[ ]
         }
      ]
    },
    "encryptionKey":{ }
}
```

After the name and description, a skillset has four main properties:

- `skills` array, an unordered collection of skills. Skills can be utilitarian (like splitting text), transformational (based on AI from Cognitive Services), or custom skills that you provide. An example of a skills array is provided in the next section.
- `cognitiveServices` is used for billable skills that call Cognitive Services APIs. Remove this section if you aren't using billable skills or Custom Entity Lookup. Attach a resource if you are.
- `knowledgeStore` (optional) specifies an Azure Storage account and settings for projecting skillset output into tables, blobs, and files in Azure Storage. Remove this section if you don't need it, otherwise specify a knowledge store.
- `encryptionKey` (optional) specifies an Azure Key Vault and customer-managed keys used to encrypt sensitive content (descriptions, connection strings, keys) in a skillset definition. Remove this property if you aren't using customer-managed encryption.

Add skills

Inside the skillset definition, the skills array specifies which skills to execute. Three to five skills are common, but you can add as many skills as necessary, subject to service limits.

The end result of an enrichment pipeline is textual content in either a search index or a knowledge store. For this reason, most skills either create text from images (OCR text, captions, tags), or analyze existing text to create new information (entities, key phrases, sentiment). Skills that operate independently are processed in parallel. Skills that depend on each other specify the output of one skill (such as key phrases) as the input of second skill (such as text translation). The search service determines the order of skill execution and the execution environment.

All skills have a type, context, inputs, and outputs. A skill might optionally have a name and description. The following example shows two unrelated built-in skills so that you can compare the basic structure.

JSON

```
"skills":[
  {
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "name": "#1",
    "description": "This skill detects organizations in the source content",
    "context": "/document",
    "categories": [ "Organization" ],
    "inputs": [
      {
        "name": "text",
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "organizations",
        "targetName": "orgs"
      }
    ]
  },
  {
      "name": "#2",
      "description": "This skill detects corporate logos in the source
files",
      "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
      "context": "/document/normalized_images/*",
      "visualFeatures": [
          "brands"
      ],
      "inputs": [
          {
              "name": "image",
              "source": "/document/normalized_images/*"
          }
      ],
      "outputs": [
          {
```

```
                "name": "brands"
            }
        ]
    }
]
```

Each skill is unique in terms of its input values and the parameters that it takes. Skill reference documentation describes all of the parameters and properties of a given skill. Although there are differences, most skills share a common set and are similarly patterned.

> **Note**
>
> You can build complex skillsets with looping and branching using the **Conditional skill** to create the expressions. The syntax is based on the **JSON Pointer** path notation, with a few modifications to identify nodes in the enrichment tree. A `"/"` traverses a level lower in the tree and `"*"` acts as a for-each operator in the context. Numerous examples in this article illustrate the **the syntax**.

Set skill context

Each skill has a context property that determines the level at which operations take place. If the "context" property isn't explicitly set, the default is `"/document"`, where the context is the whole document (the skill is called once per document).

JSON

```
"skills":[
  {
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "context": "/document",
    "inputs": [],
    "outputs": []
  },
  {
      "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
      "context": "/document/normalized_images/*",
      "visualFeatures": [],
      "inputs": [],
      "outputs": []
  }
]
```

Context is usually set to one of the following examples:

Context example	Description
"context": "/document"	(Default) Inputs and outputs are at the document level.
"context": "/document/pages/*"	Some skills like sentiment analysis perform better over smaller chunks of text. If you're splitting a large content field into pages or sentences, the context should be over each component part.
"context": "/document/normalized_images/*"	For image content, inputs and outputs are one per image in the parent document.

Context also determines where outputs are produced in the enrichment tree. For example, the Entity Recognition skill returns a property called `"organizations"`, captured as `orgs`. If the context is `"/document"`, then an "organizations" node is added as a child of `"/document"`. If you then wanted to reference this node in downstream skills, the path would be `"/document/orgs"`.

Define inputs

Skills read from and write to an enriched document. Skill inputs specify the origin of the incoming data. It's often the root node of the enriched document. For blobs, a typical skill input is the document's content property.

Skill reference documentation for each skill describes the inputs it can consume. Each input has a "name" that identifies a specific input, and a "source" that specifies the location fo the data in the enriched document. The following example is from the Entity Recognition skill:

JSON

```
"inputs": [
    {
        "name": "text",
        "source": "/document/content"
    },
    {
        "name": "languageCode",
        "source": "/document/language"
    }
]
```

- Skills can have multiple inputs. The "name" is the specific input. For Entity Recognition, the specific inputs are "text" and "languageCode".

- The "source" property specifies which field or row provides the content to be processed. For text-based skills, the source is a field in the document or row that provides text. For image-based skills, the node providing the input is normalized images.

Source example	Description
"source": "/document"	For a tabular data set, a document corresponds to a row.
"source": "/document/content"	For blobs, the source is usually the blob's content property.
"source": "/document/some-named-field"	For text-based skills, such as entity recognition or key phrase extraction, the origin should be a field that contains sufficient text to be analyzed, such as a "description" or "summary".
"source": "/document/normalized_images/*"	For image content, the source is image that's been normalized during document cracking.

If the skill iterates over an array, both context and input source should include `/*` in the correct positions.

Define outputs

Each skill is designed to emit specific kinds of output, which are referenced by name in the skillset. A skill output has a "name" and an optional "targetName".

Skill reference documentation for each skill describes the outputs it can produce. The following example is from the Entity Recognition skill:

JSON

```
"outputs": [
    {
        "name": "persons",
        "targetName": "people"
    },
    {
        "name": "organizations",
        "targetName": "orgs"
    },
    {
        "name": "locations",
        "targetName": "places"
    }
]
```

- Skills can have multiple outputs. The "name" identifies a specific output. For example, for Entity Recognition, output can be "persons", "locations", "organizations", among others.
- "targetName" specifies the name you would like this node to have in the enriched document. This is useful if skill outputs have the same name. If you have multiple skills that return the same output, use the `"targetName"` for name disambiguation in enrichment node paths. If the target name is unspecified, the name property is used for both.

Some situations call for referencing each element of an array separately. For example, suppose you want to pass *each element* of `"/document/orgs"` separately to another skill. To do so, add an asterisk to the path: `"/document/orgs/*"`.

Skill output is written to the enriched document as a new node in the enrichment tree. It might be a simple value, such as a sentiment score or language code. It could also be a collection, such as a list of organizations, people, or locations. Skill output can also be a complex structure, as is the case with the Shaper skill. The inputs of the skill determine the composition of the shape, but the output is the named object, which can be referenced in a search index, a knowledge store projection, or another skill by its name.

Add a custom skill

This section includes an example of a custom skill. The URI points to an Azure Function, which in turn invokes the model or transformation that you provide. For more information, see Define a custom interface.

Although the custom skill is executing code that is external to the pipeline, in a skills array, it's just another skill. Like the built-in skills, it has a type, context, inputs, and outputs. It also reads and writes to an enrichment tree, just as the built-in skills do. Notice that the "context" field is set to `"/document/orgs/*"` with an asterisk, meaning the enrichment step is called *for each* organization under `"/document/orgs"`.

Output, such as the company description in this example, is generated for each organization that's identified. When referring to the node in a downstream step (for example, in key phrase extraction), you would use the path `"/document/orgs/*/companyDescription"` to do so.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
```

```
  "description": "This skill calls an Azure function, which in turn calls
custom code",
  "uri": "https://indexer-e2e-webskill.azurewebsites.net/api/InvokeCode?
code=foo",
  "httpHeaders": {
      "Ocp-Apim-Subscription-Key": "foobar"
  },
  "context": "/document/orgs/*",
  "inputs": [
    {
      "name": "query",
      "source": "/document/orgs/*"
    }
  ],
  "outputs": [
    {
      "name": "description",
      "targetName": "companyDescription"
    }
  ]
}
```

Send output to a destination

Although skill output can be optionally cached for reuse purposes, it's usually temporary and exists only while skill execution is in progress.

- To send output to a field in a search index, create an output field mapping in an indexer.
- To send output to a knowledge store, create a projection.
- To send output to a downstream skill, reference the output by its node name, such as `"/document/organization"`, in the downstream skill's input source property. See Reference an annotation for examples.

Tips for a first skillset

- Assemble a representative sample of your content in Blob Storage or another supported data source and run the **Import data** wizard.

  The wizard automates several steps that can be challenging the first time around. It defines the skillset, index, and indexer, including field mappings and output field mappings. It will also define projections in a knowledge store if you're using one. For some skills, such as OCR or image analysis, the wizard adds utility skills that merge the image and text content that was separated during document cracking.

After the wizard runs, you can open each object in the Azure portal to view its JSON definition. You can also use Debug Sessions to invoke skillset execution over a target document.

- Alternatively, you can import sample Postman collections that provide a full articulation of the object definitions required to evaluate a skill.

- When you're ready to start from scratch on a new skillset, checking preliminary results is important. We recommend a search index for content validation because it's simpler to set up, in comparison to knowledge stores.

  For each skill output, define an output field mapping in the indexer, and a field in the search index.

After you run the indexer, use Search Explorer to return documents from the index and check the contents of each field to determine what the skillset detected or created.

This screenshot shows the results of an entity recognition skill that detected persons, locations, organizations, and other entities in a chunk of text. You can view the results to decide whether a skill adds value to your solution.

entity-test-idx
index

Save Discard Refresh Create Demo App Delete


```
7           "metadata_storage_name": "cGFnZS03LnBkZg2",
8           "content": "\nF\no\n\nR\ne\n\nw\no\n\nR\nd\n\nE\nA\n\nR\nT\n\nH\n\nvi\n\nForeword\n\nof all celestial bodies
within reach or view, as far as we can \n\nsee, out to the edge, the most wonderful and marvelous and \n\nmysterious is
turning out to be our own planet earth. There is \n\nnothing to match it anywhere, not yet anyway. \n\n—Lewis Thomas
\n\nSixty years ago, with the launch of Explorer 1, NASA made \n\nits first observations of Earth from space. Fifty
years ago, \n\nastronauts left Earth orbit for the first time and looked back \n\nat our “blue marble.” All of these
years later, as we send \n\nspacecraft and point our telescopes past the outer edges of  \n\nthe solar system, as we
study our planetary neighbors and  \n\nour Sun in exquisite detail, there remains much to see and \n\nexplore at home.
\n\nWe are still just getting to know Earth through the tools of \n\nscience. For centuries, painters, poets,
philosophers, and \n\nphotographers have sought to teach us something about our \n\nhome through their art.\n\n\n",
9           "persons": [
10              "Lewis Thomas"
11          ],
12          "locations": [
13              "earth",
14              "Earth",
15              "solar system",
16              "Sun",
17              "home"
18          ],
19          "organizations": [
20              "NASA"
21          ],
22          "quantities": [
23              "1",
24              "first"
25          ],
26          "dateTimes": [
27              "Sixty years ago",
28              "Fifty years ago"
29          ],
30          "urls": [],
31          "emails": [],
32          "personTypes": [
33              "astronauts",
34              "painters",
35              "poets",
36              "philosophers",
37              "photographers"
38          ],
```


Next steps

Context and input source fields are paths to nodes in an enrichment tree. As a next step, learn more about the path syntax for nodes in an enrichment tree.

Referencing annotations in a skillset

Additional resources

Documentation

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Map enrichments in indexers - Azure Cognitive Search

Export the enriched content created by a skillset by mapping its output fields to fields in a search index.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Show 5 more

Training

Module

Improve communication skills with the 21CLD skilled communication dimension - Training

This module introduces the concept of skilled communication to educators and supports them in designing learning activities to help young people develop this important skill.

Certification

Microsoft Office Specialist: 2016 Master - Certifications

Microsoft Office Specialist: 2016 Master

Debug an Azure Cognitive Search skillset in Azure portal

Article • 10/20/2022 • 7 minutes to read

Start a portal-based debug session to identify and resolve errors, validate changes, and push changes to a published skillset in your Azure Cognitive Search service.

A debug session is a cached indexer and skillset execution, scoped to a single document, that you can use to edit and test your changes interactively. If you're unfamiliar with how a debug session works, see Debug sessions in Azure Cognitive Search. To practice a debug workflow with a sample document, see Tutorial: Debug sessions.

Prerequisites

- An existing enrichment pipeline, including a data source, a skillset, an indexer, and an index.
- A **Contributor** role assignment in the Search service.
- An Azure Storage account, used to save session state.
- A **Storage Blob Data Contributor** role assignment in Azure Storage.
- If the Azure Storage account is behind a firewall, configure it to allow Search service access.

Limitations

A Debug Session works with all generally available indexer data sources and most preview data sources. The following list notes the exceptions:

- Azure Cosmos DB for MongoDB is currently not supported.
- For the Azure Cosmos DB for NoSQL, if a row fails during index and there's no corresponding metadata, the debug session might not pick the correct row.
- For the SQL API of Azure Cosmos DB, if a partitioned collection was previously non-partitioned, a Debug Session won't find the document.

Create a debug session

1. Sign in to Azure portal and find your search service.

2. In the **Overview** page of your search service, select the **Debug Sessions** tab.

3. Select **+ New Debug Session**.

4. In **Debug session name**, provide a name that will help you remember which skillset, indexer, and data source the debug session is about.

5. In **Storage connection**, find a general-purpose storage account for caching the debug session. You'll be prompted to select and optionally create a blob container in Blob Storage or Azure Data Lake Storage Gen2. You can reuse the same container for all subsequent debug sessions you create. A helpful container name might be "cognitive-search-debug-sessions".

6. In **Indexer template**, select the indexer that drives the skillset you want to debug. Copies of both the indexer and skillset are used to initialize the session.

7. In **Document to debug**, choose the first document in the index or select a specific document. If you select a specific document, depending on the data source, you'll be asked for a URI or a row ID.

   If your specific document is a blob, you'll be asked for the blob URI. You can find the URL in the blob property page in the portal.

8. Optionally, in **Indexer settings**, specify any indexer execution settings used to create the session. The settings should mimic the settings used by the actual indexer. Any indexer options that you specify in a debug session have no effect on the indexer itself.

9. Your configuration should look similar to this screenshot. Select **Save Session** to get started.

The debug session begins by executing the indexer and skillset on the selected document. The document's content and metadata created will be visible and available in the session.

A debug session can be canceled while it's executing using the **Cancel** button. If you hit the **Cancel** button you should be able to analyze partial results.

It is expected for a debug session to take longer to execute than the indexer since it goes through extra processing.

Start with errors and warnings

Indexer execution history in the portal gives you the full error and warning list for all documents. In a debug session, the errors and warnings will be limited to one document. You'll work through this list, make your changes, and then return to the list to verify whether issues are resolved.

To view the messages, select a skill in **AI Enrichment** > **Skill Graph** and then select **Errors/Warnings** in the details pane.

As a best practice, resolve problems with inputs before moving on to outputs.

To prove whether a modification resolves an error, follow these steps:

1. Select **Save** in the skill details pane to preserve your changes.
2. Select **Run** in the session window to invoke skillset execution using the modified definition.
3. Return to **Errors/Warnings** to see if the count is reduced. The list won't be refreshed until you open the tab.

View content of enrichment nodes

AI enrichment pipelines extract or infer information and structure from source documents, creating an enriched document in the process. An enriched document is first created during document cracking and populated with a root node (`/document`), plus nodes for any content that is lifted directly from the data source, such as metadata and the document key. More nodes are created by skills during skill execution, where each skill output adds a new node to the enrichment tree.

Enriched documents are internal, but a debug session gives you access to the content produced during skill execution. To view the content or output of each skill, follow these steps:

1. Start with the default views: **AI enrichment** > **Skill Graph**, with the graph type set to **Dependency Graph**.
2. Select a skill.

3. In the details pane to the right, select **Executions**, select an OUTPUT, and then open the Expression Evaluator (`</>`) to view the expression and its result.

4. Alternatively, open **AI enrichment** > **Enriched Data Structure** to scroll down the list of nodes. The list includes potential and actual nodes, with a column for output, and another column that indicates the upstream object used to produce the output.

Edit skill definitions

If the field mappings are correct, check individual skills for configuration and content. If a skill fails to produce output, it might be missing a property or parameter, which can be determined through error and validation messages.

Other issues, such as an invalid context or input expression, can be harder to resolve because the error will tell you what is wrong, but not how to fix it. For help with context and input syntax, see Reference annotations in an Azure Cognitive Search skillset. For help with individual messages, see Troubleshooting common indexer errors and warnings.

The following steps show you how to get information about a skill.

1. In **AI enrichment** > **Skill Graph**, select a skill. The Skill Details pane opens to the right.

2. Edit a skill definition using either approach:

   - **Skill Settings** if you prefer a visual editor
   - **Skill JSON Editor** to edit the JSON document directly

3. Check the path syntax for referencing nodes in an enrichment tree. Following are some of the most common input paths:

   - `/document/content` for chunks of text. This node is populated from the blob's content property.
   - `/document/merged_content` for chunks of text in skillets that include Text Merge skill.
   - `/document/normalized_images/*` for text that is recognized or inferred from images.

Check field mappings

If skills produce output but the search index is empty, check the field mappings. Field mappings specify how content moves out of the pipeline and into a search index.

1. Start with the default views: **AI enrichment** > **Skill Graph**, with the graph type set to **Dependency Graph**.

2. Select **Field Mappings** near the top. You should find at least the document key that uniquely identifies and associates each search document in the search index with its source document in the data source.

   If you're importing raw content straight from the data source, bypassing enrichment, you should find those fields in **Field Mappings**.

3. Select **Output Field Mappings** at the bottom of the graph. Here you'll find mappings from skill outputs to target fields in the search index. Unless you used the Import Data wizard, output field mappings are defined manually and could be incomplete or mistyped.

   Verify that the fields in **Output Field Mappings** exist in the search index as specified, checking for spelling and enrichment node path syntax.

Debug a custom skill locally

Custom skills can be more challenging to debug because the code runs externally, so the debug session can't be used to debug them. This section describes how to locally debug your Custom Web API skill, debug session, Visual Studio Code and ngrok. This technique works with custom skills that execute in Azure Functions or any other Web Framework that runs locally (for example, FastAPI).

Run ngrok

ngrok is a cross-platform application that can create a tunneling or forwarding URL, so that internet requests reach your local machine. Use ngrok to forward requests from an enrichment pipeline in your search service to your machine to allow local debugging.

1. Install ngrok.

2. Open a terminal and go to the folder with the ngrok executable.

3. Run ngrok with the following command to create a new tunnel:

Console

```
ngrok http 7071
```

> **Note**
>
> By default, Azure functions are exposed on 7071. Other tools and configurations might require that you provide a different port.

4. When ngrok starts, copy and save the public forwarding URL for the next step. The forwarding URL is randomly generated.

```
Session Status                online
Account                       [redacted]
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://3b7e-2a0e-41b-8f8f-0-90c8-fd24-9bd7-22b6.ngrok.io -> http://localhost:7071
Forwarding                    https://3b7e-2a0e-41b-8f8f-0-90c8-fd24-9bd7-22b6.ngrok.io -> http://localhost:7071

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Configure in Azure portal

Within the debug session, modify your Custom Web API Skill URI to call the ngrok forwarding URL. Ensure that you append "/api/FunctionName" when using Azure Function for executing the skillset code.

You can edit the skill definition in the portal.

Test your code

At this point, new requests from your debug session should now be sent to your local Azure Function. You can use breakpoints in your Visual Studio code to debug your code or run step by step.

Next steps

Now that you understand the layout and capabilities of the Debug Sessions visual editor, try the tutorial for a hands-on experience.

Tutorial: Explore Debug sessions

Additional resources

Documentation

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Create a skillset - Azure Cognitive Search

A skillset defines data extraction, natural language processing, and image analysis steps. A skillset is attached to indexer. It's used to enrich and extract information from source data for use in Azure Cognitive Search.

Map enrichments in indexers - Azure Cognitive Search

Export the enriched content created by a skillset by mapping its output fields to fields in a search index.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Text Merge cognitive skill - Azure Cognitive Search

Merge text from a collection of fields into one consolidated field. Use this cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Show 5 more

Reference an annotation in an Azure Cognitive Search skillset

Article • 10/12/2022 • 6 minutes to read

In this article, you'll learn how to reference *annotations* (or an enrichment node) in skill definitions, using examples to illustrate various scenarios.

Skills read inputs and write outputs to nodes in an enriched document tree, building the tree as the enrichments progress. Any node can be referenced in an input for further downstream enrichment, or mapped to an output field in an index. This article introduces the syntax and provides examples for specifying a path to a node. For the full syntax, see Skill context and input annotation language language.

Paths to an annotation are specified in the "context" and "source" properties of a skillset, and in output field mappings in an indexer. Here's an example of what paths might look like in a skillset:

```
{
  "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",
  "name": "#2",
  "description": null,
  "context": "/document/HotelId",
  "defaultFromLanguageCode": null,
  "defaultToLanguageCode": "es",
  "suggestedFrom": "en",
  "inputs": [
    {
      "name": "text",
      "source": "/document/Description"
    }
  ],
  "outputs": [
    {
      "name": "translatedText",
      "targetName": "translated_text"
    }
  ]
}
```

The example in the screenshot illustrates the path for an item in an Azure Cosmos DB collection.

- "context" path is `/document/HotelId` because the collection is partitioned into documents by the `/HotelId` field.

- "source" path is `/document/Description` because the skill is a translation skill, and the field that you'll want the skill to translate is the `Description` field in each document.

Background concepts

Before reviewing the syntax, let's revisit a few important concepts to better understand the examples provided later in this article.

Term	Description
"enriched document"	An enriched document is an internal structure that collects skill output as it's created and it holds all annotations related to a document. Think of an enriched document as a tree of annotations. Generally, an annotation created from a previous annotation becomes its child. Enriched documents only exist for the duration of skillset execution. Once content is mapped to the search index, the enriched document is no longer needed. Although you don't interact with enriched documents directly, it's useful to have a mental model of the documents when creating a skillset.
"annotation"	Within an enriched document, a node that is created and populated by a skill, such as "text" and "layoutText" in the OCR skill, is called an annotation. An enriched document is populated with both annotations and unchanged field values or metadata copied from the source.
"context"	The scope of enrichment, which is either the entire document, a portion of a document, or if you're working with images, the extracted images from a document. By default, the enrichment context is at the `"/document"` level, scoped to individual documents contained in the data source. When a skill runs, the outputs of that skill become properties of the defined context.

Paths for different scenarios

Paths are specified in the "context" and "source" properties of a skillset, and in the output field mappings in an indexer.

All paths start with `/document`. An enriched document is created in the "document cracking" stage of indexer execution, when the indexer opens a document or reads in a row from the data source. Initially, the only node in an enriched document is the root node (/document), and it's the node from which all other enrichments occur.

The following list includes several common examples:

- `/document` is the root node and indicates an entire blob in Azure Storage, or a row in a SQL table.
- `/document/{key}` is the syntax for a document or item in an Azure Cosmos DB collection, where `{key}` is the actual key, such as `/document/HotelId` in the previous example.
- `/document/content` specifies the "content" property of a JSON blob.
- `/document/{field}` is the syntax for an operation performed on a specific field, such as translating the `/document/Description` field, seen in the previous example.
- `/document/pages/*` or `/document/sentences/*` become the context if you're breaking a large document into smaller chunks for processing. If "context" is `/document/pages/*`, the skill executes once over each page in the document. Because there might be more than one page or sentence, you'll append `/*` to catch them all.
- `/document/normalized_images/*` is created during document cracking if the document contains images. All paths to images start with normalized_images. Since there are often multiple images embedded in a document, append `/*`.

Examples in the remainder of this article are based on the "content" field generated automatically by Azure Blob indexers as part of the document cracking phase. When referring to documents from a Blob container, use a format such as `"/document/content"`, where the "content" field is part of the "document".

Example 1: Simple annotation reference

In Azure Blob Storage, suppose you have a variety of files containing references to people's names that you want to extract using entity recognition. In the skill definition below, `"/document/content"` is the textual representation of the entire document, and "people" is an extraction of full names for entities identified as persons.

Because the default context is `"/document"`, the list of people can now be referenced as `"/document/people"`. In this specific case `"/document/people"` is an annotation, which could now be mapped to a field in an index, or used in another skill in the same skillset.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "categories": [ "Person"],
  "defaultLanguageCode": "en",
  "inputs": [
    {
      "name": "text",
```

```
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "persons",
        "targetName": "people"
      }
    ]
  }
```

Example 2: Reference an array within a document

This example builds on the previous one, showing you how to invoke an enrichment step multiple times over the same document. Assume the previous example generated an array of strings with 10 people names from a single document. A reasonable next step might be a second enrichment that extracts the last name from a full name. Because there are 10 names, you want this step to be called 10 times in this document, once for each person.

To invoke the right number of iterations, set the context as `"/document/people/*"`, where the asterisk (`"*"`) represents all the nodes in the enriched document as descendants of `"/document/people"`. Although this skill is only defined once in the skills array, it's called for each member within the document until all members are processed.

JSON

```
  {
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
    "description": "Fictitious skill that gets the last name from a full
name",
    "uri": "http://names.azurewebsites.net/api/GetLastName",
    "context" : "/document/people/*",
    "defaultLanguageCode": "en",
    "inputs": [
      {
        "name": "fullname",
        "source": "/document/people/*"
      }
    ],
    "outputs": [
      {
        "name": "lastname",
        "targetName": "last"
      }
```

```
    ]
  }
```

When annotations are arrays or collections of strings, you might want to target specific members rather than the array as a whole. The above example generates an annotation called `"last"` under each node represented by the context. If you want to refer to this family of annotations, you could use the syntax `"/document/people/*/last"`. If you want to refer to a particular annotation, you could use an explicit index: `"/document/people/1/last"` to reference the last name of the first person identified in the document. Notice that in this syntax arrays are "0 indexed".

Example 3: Reference members within an array

Sometimes you need to group all annotations of a particular type to pass them to a particular skill. Consider a hypothetical custom skill that identifies the most common last name from all the last names extracted in Example 2. To provide just the last names to the custom skill, specify the context as `"/document"` and the input as `"/document/people/*/lastname"`.

Notice that the cardinality of `"/document/people/*/lastname"` is larger than that of document. There may be 10 lastname nodes while there's only one document node for this document. In that case, the system will automatically create an array of `"/document/people/*/lastname"` containing all of the elements in the document.

JSON

```
  {
    "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
    "description": "Fictitious skill that gets the most common string from
an array of strings",
    "uri": "http://names.azurewebsites.net/api/MostCommonString",
    "context" : "/document",
    "inputs": [
      {
        "name": "strings",
        "source": "/document/people/*/lastname"
      }
    ],
    "outputs": [
      {
        "name": "mostcommon",
        "targetName": "common-lastname"
      }
    ]
  }
```

Tips for annotation path troubleshooting

If you're having trouble with specifying skill inputs, these tips might help you move forward:

- Run the Import data wizard over your data to review the skillset definitions and field mappings that the wizard generates.
- Start a debug session on a skillset to view the structure of an enriched document. You can edit the paths and other parts of the skill definition, and then run the skill to validate your changes.

See also

- Skill context and input annotation language
- How to integrate a custom skill into an enrichment pipeline
- How to define a skillset
- Create Skillset (REST)
- How to map enriched fields to an index

Map enriched output to fields in a search index in Azure Cognitive Search

Article • 10/12/2022 • 6 minutes to read

This article explains how to set up *output field mappings* that determine a data path between in-memory data structures created during skill processing, and target fields in a search index. An output field mapping is defined in an indexer and has the following elements:

JSON

```
"outputFieldMappings": [
  {
    "sourceFieldName": "document/path-to-a-node-in-an-enriched-document",
    "targetFieldName": "some-search-field-in-an-index",
    "mappingFunction": null
  }
],
```

In contrast with a fieldMappings definition that maps a path between two physical data structures, an `outputFieldMappings` definition maps in-memory data to fields in a search index.

Output field mappings are required if your indexer has an attached skillset that creates new information, such as text translation or key phrase extraction. During indexer execution, AI-generated information exists in memory only. To persist this information in a search index, you'll need to tell the indexer where to send the data.

Output field mappings can also be used to retrieve specific nodes in a source document's complex type. For example, you might want just "FullName/LastName" in a multi-part "FullName" property. When you don't need the full complex structure, you can flatten individual nodes in a nested data structures, and then use an output field mapping to send the output to a string collection in your search index.

Output field mappings apply to:

- In-memory content that's created by skills or extracted by an indexer. The source field is a node in an enriched document tree.

- Search indexes. If you're populating a knowledge store, use projections for data path configuration.

Output field mappings are applied after skillset execution or after document cracking if there's no associated skillset.

Define an output field mapping

Output field mappings are added to the `outputFieldMappings` array in an indexer definition, typically placed after the `fieldMappings` array. An output field mapping consists of three parts.

JSON

```
"fieldMappings": []
"outputFieldMappings": [
  {
    "sourceFieldName": "/document/path-to-a-node-in-an-enriched-document",
    "targetFieldName": "some-search-field-in-an-index",
    "mappingFunction": null
  }
],
```

Property	Description
sourceFieldName	Required. Specifies a path to enriched content. An example might be `/document/content`. See Reference annotations in an Azure Cognitive Search skillset for path syntax and examples.
targetFieldName	Optional. Specifies the search field that receives the enriched content. Target fields must be top-level simple fields or collections. It can't be a path to a subfield in a complex type. If you want to retrieve specific nodes in a complex structure, you can flatten individual nodes in memory, and then send the output to a string collection in your index.
mappingFunction	Optional. Adds extra processing provided by mapping functions supported by indexers. In the case of enrichment nodes, encoding and decoding are the most commonly used functions.

You can use the REST API or an Azure SDK to define output field mappings.

> **Tip**
>
> Indexers created by the **Import data wizard** include output field mappings generated by the wizard. If you need examples, run the wizard over your data

source to see the rendered definition.

REST APIs

Use Create Indexer (REST) or Update Indexer (REST), any API version.

This example adds entities and sentiment labels extracted from a blob's content property to fields in a search index.

JSON

```
PUT https://[service name].search.windows.net/indexers/myindexer?api-version=[api-version]
Content-Type: application/json
api-key: [admin key]
{
    "name": "myIndexer",
    "dataSourceName": "myDataSource",
    "targetIndexName": "myIndex",
    "skillsetName": "myFirstSkillSet",
    "fieldMappings": [],
    "outputFieldMappings": [
        {
            "sourceFieldName": "/document/content/organizations/*/description",
            "targetFieldName": "descriptions",
            "mappingFunction": {
                "name": "base64Decode"
            }
        },
        {
            "sourceFieldName": "/document/content/organizations",
            "targetFieldName": "orgNames"
        },
        {
            "sourceFieldName": "/document/content/sentiment",
            "targetFieldName": "sentiment"
        }
    ]
}
```

For each output field mapping, set the location of the data in the enriched document tree (sourceFieldName), and the name of the field as referenced in the index (targetFieldName). Assign any mapping functions that you require to transform the content of a field before it's stored in the index.

Flatten complex structures into a string collection

If your source data is composed of nested or hierarchical JSON, you can't use field mappings to set up the data paths. Instead, your search index must mirror the source data structure for at each level for a full import.

This section walks you through an import process that produces a one-to-one reflection of a complex document on both the source and target sides. Next, it uses the same source document to illustrate the retrieval and flattening of individual nodes into string collections.

Here's an example of a document in Azure Cosmos DB with nested JSON:

JSON

```
{
   "palette":"primary colors",
   "colors":[
      {
         "name":"blue",
         "medium":[
            "acrylic",
            "oil",
            "pastel"
         ]
      },
      {
         "name":"red",
         "medium":[
            "acrylic",
            "pastel",
            "watercolor"
         ]
      },
      {
         "name":"yellow",
         "medium":[
            "acrylic",
            "watercolor"
         ]
      }
   ]
}
```

If you wanted to fully index the above source document, you'd create an index definition where the field names, levels, and types are reflected as a complex type. Because field

mappings aren't supported for complex types in the search index, your index definition must mirror the source document.

JSON

```
{
  "name": "my-test-index",
  "defaultScoringProfile": "",
  "fields": [
    { "name": "id", "type": "Edm.String", "searchable": false,
"retrievable": true, "key": true},
    { "name": "palette", "type": "Edm.String", "searchable": true,
"retrievable": true },
    { "name": "colors", "type": "Collection(Edm.ComplexType)",
      "fields": [
        {
          "name": "name",
          "type": "Edm.String",
          "searchable": true,
          "retrievable": true
        },
        {
          "name": "medium",
          "type": "Collection(Edm.String)",
          "searchable": true,
          "retrievable": true,
        }
      ]
    }
  ]
}
```

Here's a sample indexer definition that executes the import (notice there are no field mappings and no skillset).

JSON

```
{
  "name": "my-test-indexer",
  "dataSourceName": "my-test-ds",
  "skillsetName": null,
  "targetIndexName": "my-test-index",

  "fieldMappings": [],
  "outputFieldMappings": []
}
```

The result is the following sample search document, similar to the original in Azure Cosmos DB.

JSON

```
{
  "value": [
    {
      "@search.score": 1,
      "id": "240a98f5-90c9-406b-a8c8-f50ff86f116c",
      "palette": "primary colors",
      "colors": [
        {
          "name": "blue",
          "medium": [
            "acrylic",
            "oil",
            "pastel"
          ]
        },
        {
          "name": "red",
          "medium": [
            "acrylic",
            "pastel",
            "watercolor"
          ]
        },
        {
          "name": "yellow",
          "medium": [
            "acrylic",
            "watercolor"
          ]
        }
      ]
    }
  ]
}
```

An alternative rendering in a search index is to flatten individual nodes in the source's nested structure into a string collection in a search index.

To accomplish this task, you'll need an `outputFieldMapping` that maps an in-memory node to a string collection in the index. Although output field mappings primarily apply to skill outputs, you can also use them to address nodes after "document cracking" where the indexer opens a source document and reads it into memory.

Below is a sample index definition in Cognitive Search, using string collections to receive flattened output:

JSON

```
{
  "name": "my-new-flattened-index",
  "defaultScoringProfile": "",
  "fields": [
    { "name": "id", "type": "Edm.String", "searchable": false,
"retrievable": true, "key": true },
    { "name": "palette", "type": "Edm.String", "searchable": true,
"retrievable": true },
    { "name": "color_names", "type": "Collection(Edm.String)", "searchable":
true, "retrievable": true },
    { "name": "color_mediums", "type": "Collection(Edm.String)",
"searchable": true, "retrievable": true}
  ]
}
```

Here's the sample indexer definition, using `outputFieldMappings` to associate the nested JSON with the string collection fields. Notice that the source field uses the path syntax for enrichment nodes, even though there's no skillset. Enriched documents are created in the system during document cracking, which means you can access nodes in each document tree as long as those nodes exist when the document is cracked.

JSON

```
{
  "name": "my-test-indexer",
  "dataSourceName": "my-test-ds",
  "skillsetName": null,
  "targetIndexName": "my-new-flattened-index",
  "parameters": {  },
  "fieldMappings": [   ],
  "outputFieldMappings": [
    {
       "sourceFieldName": "/document/colors/*/name",
       "targetFieldName": "color_names"
    },
    {
       "sourceFieldName": "/document/colors/*/medium",
       "targetFieldName": "color_mediums"
    }
  ]
}
```

Results from the above definition are as follows. Simplifying the structure loses context in this case. There's no longer any associations between a given color and the mediums it's available in. However, depending on your scenario, a result similar to the one shown below might be exactly what you need.

JSON

```
{
  "value": [
    {
      "@search.score": 1,
      "id": "240a98f5-90c9-406b-a8c8-f50ff86f116c",
      "palette": "primary colors",
      "color_names": [
        "blue",
        "red",
        "yellow"
      ],
      "color_mediums": [
        "[\"acrylic\",\"oil\",\"pastel\"]",
        "[\"acrylic\",\"pastel\",\"watercolor\"]",
        "[\"acrylic\",\"watercolor\"]"
      ]
    }
  ]
}
```


See also

- Define field mappings in a search indexer
- AI enrichment overview
- Skillset overview

Additional resources

🕮 Documentation

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Create a skillset - Azure Cognitive Search

A skillset defines data extraction, natural language processing, and image analysis steps. A skillset is attached to indexer. It's used to enrich and extract information from source data for use in Azure Cognitive Search.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Tutorial: create a custom analyzer - Azure Cognitive Search

Learn how to build a custom analyzer to improve the quality of search results in Azure Cognitive Search.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Show 5 more

Extract text and information from images in AI enrichment

Article • 08/30/2022 • 19 minutes to read

Through AI enrichment, Azure Cognitive Search gives you several options for creating and extracting searchable text from images, including:

- OCR for optical character recognition of text and digits
- Image Analysis that describes images through visual features
- Custom skills to invoke any external image processing that you want to provide

Through OCR, you can extract text from photos or pictures containing alphanumeric text, such as the word "STOP" in a stop sign. Through image analysis, you can generate a text representation of an image, such as "dandelion" for a photo of a dandelion, or the color "yellow". You can also extract metadata about the image, such as its size.

This article covers the fundamentals of working with images, and also describes several common scenarios, such as working with embedded images, custom skills, and overlaying visualizations on original images.

To work with image content in a skillset, you'll need:

- ✓ Source files that include images
- ✓ A search indexer, configured for image actions
- ✓ A skillset with built-in or custom skills that invoke OCR or image analysis
- ✓ A search index with fields to receive the analyzed text output, plus output field mappings in the indexer that establish association.

Optionally, you can define projections to accept image-analyzed output into a knowledge store for data mining scenarios.

Set up source files

Image processing is indexer-driven, which means that the raw inputs must be in a supported data source.

- Image analysis supports JPEG, PNG, GIF, and BMP
- OCR supports JPEG, PNG, BMP, and TIF

Images are either standalone binary files or embedded in documents (PDF, RTF, and Microsoft application files). A maximum of 1000 images will be extracted from a given

document. If there are more than 1000 images in a document, the first 1000 will be extracted and a warning will be generated.

Azure Blob Storage is the most frequently used storage for image processing in Cognitive Search. There are three main tasks related to retrieving images from a blob container:

- Enable access to content in the container. If you're using a full access connection string that includes a key, the key gives you permission to the content. Alternatively, you can authenticate using Azure Active Directory (Azure AD) or connect as a trusted service.
- Create a data source of type "azureblob" that connects to the blob container storing your files.
- Review service tier limits to make sure that your source data is under maximum size and quantity limits for indexers and enrichment.

Configure indexers for image processing

Extracting images from the source content files is the first step of indexer processing. Extracted images are queued for image processing. Extracted text is queued for text processing, if applicable.

Image processing requires image normalization to make images more uniform for downstream processing. This second step occurs automatically and is internal to indexer processing. As a developer, you enable image normalization by setting the `"imageAction"` parameter in indexer configuration.

Image normalization includes the following operations:

- Large images are resized to a maximum height and width to make them uniform and consumable during skillset processing.
- For images that have metadata on orientation, image rotation is adjusted for vertical loading.

Metadata adjustments are captured in a complex type created for each image. You cannot turn off image normalization. Skills that iterate over images, such as OCR and image analysis, expect normalized images.

1. Create or Update an indexer to set the configuration properties:

```
JSON
```

```
{
  "parameters":
  {
    "configuration":
    {
       "dataToExtract": "contentAndMetadata",
       "parsingMode": "default",
       "imageAction": "generateNormalizedImages"
    }
  }
}
```

2. Set `"dataToExtract"` to `"contentAndMetadata"` (required).

3. Verify that the `"parsingMode"` is set to default (required).

   This parameter determines the granularity of search documents created in the index. The default mode sets up a one-to-one correspondence so that one blob results in one search document. If documents are large, or if skills require smaller chunks of text, you can add Text Split skill that subdivides a document into paging for processing purposes. But for search scenarios, one blob per document is required if enrichment includes image processing.

4. Set `"imageAction"` to enable the *normalized_images* node in an enrichment tree (required):

   - `"generateNormalizedImages"` to generate an array of normalized images as part of document cracking.

   - `"generateNormalizedImagePerPage"` (applies to PDF only) to generate an array of normalized images where each page in the PDF is rendered to one output image. For non-PDF files, the behavior of this parameter is similar as if you had set "generateNormalizedImages". However, note that setting "generateNormalizedImagePerPage" can make indexing operation less performant by design (especially for big documents) since several images would have to be generated.

5. Optionally, adjust the width or height of the generated normalized images:

   - `"normalizedImageMaxWidth"` (in pixels). Default is 2000. Maximum value is 10000.

   - `"normalizedImageMaxHeight"` (in pixels). Default is 2000. Maximum value is 10000.

The default of 2000 pixels for the normalized images maximum width and height is based on the maximum sizes supported by the OCR skill and the image analysis skill. The OCR skill supports a maximum width and height of 4200 for non-English languages, and 10000 for English. If you increase the maximum limits, processing could fail on larger images depending on your skillset definition and the language of the documents.

- Optionally, set file type criteria if the workload targets a specific file type. Blob indexer configuration includes file inclusion and exclusion settings. You can filter out files you don't want.

JSON

```
{
  "parameters" : {
      "configuration" : {
          "indexedFileNameExtensions" : ".pdf, .docx",
          "excludedFileNameExtensions" : ".png, .jpeg"
      }
  }
}
```

About normalized images

When "imageAction" is set to a value other than "none", the new *normalized_images* field will contain an array of images. Each image is a complex type that has the following members:

Image member	Description
data	BASE64 encoded string of the normalized image in JPEG format.
width	Width of the normalized image in pixels.
height	Height of the normalized image in pixels.
originalWidth	The original width of the image before normalization.
originalHeight	The original height of the image before normalization.
rotationFromOriginal	Counter-clockwise rotation in degrees that occurred to create the normalized image. A value between 0 degrees and 360 degrees. This step reads the metadata from the image that is generated by a camera or scanner. Usually a multiple of 90 degrees.

Image member	Description
contentOffset	The character offset within the content field where the image was extracted from. This field is only applicable for files with embedded images. Note that the *contentOffset* for images extracted from PDF documents will always be at the end of the text on the page it was extracted from in the document. This means images will be after all the text on that page, regardless of the original location of the image in the page.
pageNumber	If the image was extracted or rendered from a PDF, this field contains the page number in the PDF it was extracted or rendered from, starting from 1. If the image was not from a PDF, this field will be 0.

Sample value of *normalized_images*:

JSON

```
[
  {
    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",
    "width": 500,
    "height": 300,
    "originalWidth": 5000,
    "originalHeight": 3000,
    "rotationFromOriginal": 90,
    "contentOffset": 500,
    "pageNumber": 2
  }
]
```

Define skillsets for image processing

This section supplements the skill reference articles by providing context for working with skill inputs, outputs, and patterns, as they relate to image processing.

1. Create or update a skillset to add skills.

2. Add templates for OCR and Image Analysis from the portal, or copy the definitions from the skill reference documentation. Insert them into the skills array of your skillset definition.

3. If necessary, include multi-service key in the Cognitive Services property of the skillset. Cognitive Search makes calls to a billable Azure Cognitive Services resource for OCR and image analysis for transactions that exceed the free limit (20 per

indexer per day). Cognitive Services must be in the same region as your search service.

4. If original images are embedded in PDF or application files like PPTX or DOCX, you'll need to add a Text Merge skill if you want image output and text output together. Working with embedded images is discussed further on in this article.

Once the basic framework of your skillset is created and Cognitive Services is configured, you can focus on each individual image skill, defining inputs and source context, and mapping outputs to fields in either an index or knowledge store.

> **Note**
>
> See **REST Tutorial: Use REST and AI to generate searchable content from Azure blobs** for an example skillset that combines image processing with downstream natural language processing. It shows how to feed skill imaging output into entity recognition and key phrase extraction.

About inputs for image processing

As noted, images are extracted during document cracking and then normalized as a preliminary step. The normalized images are the inputs to any image processing skill, and are always represented in an enriched document tree in either one of two ways:

- `"/document/normalized_images/*"` is for documents that are processed whole.
- `"/document/normalized_images/*/pages"` is for documents that are processed in chunks (pages).

Whether you're using OCR and image analysis in the same, inputs have virtually the same construction:

JSON

```
  {
    "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
    "context": "/document/normalized_images/*",
    "detectOrientation": true,
    "inputs": [
      {
        "name": "image",
        "source": "/document/normalized_images/*"
      }
    ],
    "outputs": [ ]
```

```
    },
    {
      "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
      "context": "/document/normalized_images/*",
      "visualFeatures": [ "tags", "description" ],
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [ ]
    }
```

Map outputs to search fields

Cognitive Search is a full text search and knowledge mining solution, so Image Analysis and OCR skill output is always text. Output text is represented as nodes in an internal enriched document tree, and each node must be mapped to fields in a search index or projections in a knowledge store to make the content available in your app.

1. In the skillset, review the "outputs" section of each skill to determine which nodes exist in the enriched document:

   JSON

   ```
   {
     "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
     "context": "/document/normalized_images/*",
     "detectOrientation": true,
     "inputs": [ ],
     "outputs": [
       {
         "name": "text",
         "targetName": "text"
       },
       {
         "name": "layoutText",
         "targetName": "layoutText"
       }
     ]
   }
   ```

2. Create or update a search index to add fields to accept the skill outputs.

   In the fields collection below, "content" is blob content. "Metadata_storage_name" contains the name of the file (make sure it is "retrievable"). "Metadata_storage_path" is the unique path of the blob and is the default

document key. "Merged_content" is output from Text Merge (useful when images are embedded).

"Text" and "layoutText" are OCR skill outputs and must be a string collection in order to the capture all of the OCR-generated output for the entire document.

JSON

```
  "fields": [
    {
      "name": "content",
      "type": "Edm.String",
      "filterable": false,
      "retrievable": true,
      "searchable": true,
      "sortable": false
    },
    {
      "name": "metadata_storage_name",
      "type": "Edm.String",
      "filterable": true,
      "retrievable": true,
      "searchable": true,
      "sortable": false
    },
    {
      "name": "metadata_storage_path",
      "type": "Edm.String",
      "filterable": false,
      "key": true,
      "retrievable": true,
      "searchable": false,
      "sortable": false
    },
    {
      "name": "merged_content",
      "type": "Edm.String",
      "filterable": false,
      "retrievable": true,
      "searchable": true,
      "sortable": false
    },
    {
      "name": "text",
      "type": "Collection(Edm.String)",
      "filterable": false,
      "retrievable": true,
      "searchable": true
    },
    {
      "name": "layoutText",
      "type": "Collection(Edm.String)",
      "filterable": false,
```

```
      "retrievable": true,
      "searchable": true
    }
  ],
```

3. Update the indexer to map skillset output (nodes in an enrichment tree) to index fields.

   Enriched documents are internal. To "externalize" the nodes, an output field mapping specifies the data structure that receives node content. This is the data that will be accessed by your app. Below is an example of a "text" node (OCR output) mapped to a "text" field in a search index.

   JSON

   ```
   "outputFieldMappings": [
     {
       "sourceFieldName": "/document/normalized_images/*/text",
       "targetFieldName": "text"
     },
     {
       "sourceFieldName": "/document/normalized_images/*/layoutText",
       "targetFieldName": "layoutText"
     }
   ]
   ```

4. Run the indexer to invoke source document retrieval, image processing, and indexing.

Verify results

Run a query against the index to check the results of image processing. Use Search Explorer as a search client, or any tool that sends HTTP requests. The following query selects fields that contain the output of image processing.

HTTP

```
POST /indexes/[index name]/docs/search?api-version=[api-version]
{
    "search": "*",
    "select": "metadata_storage_name, text, layoutText, imageCaption,
imageTags"
}
```

OCR recognizes text in image files. This means that OCR fields ("text" and "layoutText") will be empty if source documents are pure text or pure imagery. Similarly, image

analysis fields ("imageCaption" and "imageTags") will be empty if source document inputs are strictly text. Indexer execution will emit warnings if imaging inputs are empty. Such warnings are to be expected when nodes are unpopulated in the enriched document. Recall that blob indexing lets you include or exclude file types if you want to work with content types in isolation. You can use these setting to reduce noise during indexer runs.

An alternate query for checking results might include the "content" and "merged_content" fields. Notice that those fields will include content for any blob file, even those where there was no image processing performed.

About skill outputs

Skill outputs include "text" (OCR), "layoutText" (OCR), "merged_content", "captions" (image analysis), "tags" (image analysis):

- "text" stores OCR-generated output. This node should be mapped to field of type `Collection(Edm.String)`. There is one "text" field per search document consisting of comma-delimited strings for documents that contain multiple images. The following illustration shows OCR output for three documents. First is a document containing a file with no images. Second is a document (image file) containing one word, "Microsoft". Third is a document containing multiple images, some without any text (`"",`).

JSON

```
"value": [
    {
        "@search.score": 1,
        "metadata_storage_name": "facts-about-microsoft.html",
        "text": []
    },
    {
        "@search.score": 1,
        "metadata_storage_name": "guthrie.jpg",
        "text": [ "Microsoft" ]
    },
    {
        "@search.score": 1,
        "metadata_storage_name": "Cognitive Services and Content
Intelligence.pptx",
        "text": [
            "",
            "Microsoft",
            "",
            "",
            "",
```

```
            "Cognitive Search and Augmentation Combining Microsoft
Cognitive Services and Azure Search"
        ]
    }
]
```

- "layoutText" stores OCR-generated information about text location on the page, described in terms of bounding boxes and coordinates of the normalized image. This node should be mapped to field of type `Collection(Edm.String)`. There is one "layoutText" field per search document consisting of comma-delimited strings.
- "merged_content" stores the output of a Text Merge skill, and it should be one large field of type `Edm.String` that contains raw text from the source document, with embedded "text" in place of an image. If files are text-only, then OCR and image analysis have nothing to do, and "merged_content" will be the same as "content" (a blob property that contains the content of the blob).
- "imageCaption" captures a description of an image as individuals tags and a longer text description.
- "imageTags" stores tags about an image as a collection of keywords, one collection for all images in the source document.

The following screenshot is an illustration of a PDF that includes text and embedded images. Document cracking detected three embedded images: flock of seagulls, map, eagle. Other text in the example (including titles, headings, and body text) was extracted as text and excluded from image processing.

Image analysis output is illustrated in the JSON below (search result). The skill definition allows you to specify which visual features are of interest. For this example, tags and descriptions were produced, but there are more outputs to choose from.

- "imageCaption" output is an array of descriptions, one per image, denoted by "tags" consisting of single words and longer phrases that describe the image. Notice the tags consisting of "a flock of seagulls are swimming in the water", or "a close up of a bird".

- "imageTags" output is an array of single tags, listed in the order of creation. Notice that tags will repeat. There is no aggregation or grouping.

JSON


```
 "imageCaption": [
      "{\"tags\":
[\"bird\",\"outdoor\",\"water\",\"flock\",\"many\",\"lot\",\"bunch\",\"group
\",\"several\",\"gathered\",\"pond\",\"lake\",\"different\",\"family\",\"fly
ing\",\"standing\",\"little\",\"air\",\"beach\",\"swimming\",\"large\",\"dog
\",\"landing\",\"jumping\",\"playing\"],\"captions\":[{\"text\":\"a flock of
seagulls are swimming in the water\",\"confidence\":0.70419257326275686}]}",
      "{\"tags\":[\"map\"],\"captions\":
[{\"text\":\"map\",\"confidence\":0.99942880868911743}]}",
      "{\"tags\":
[\"animal\",\"bird\",\"raptor\",\"eagle\",\"sitting\",\"table\"],\"captions\
":[{\"text\":\"a close up of a
bird\",\"confidence\":0.89643581933539462}]}",
    . . .

 "imageTags": [
```

```
      "bird",
      "outdoor",
      "water",
      "flock",
      "animal",
      "bunch",
      "group",
      "several",
      "drink",
      "gathered",
      "pond",
      "different",
      "family",
      "same",
      "map",
      "text",
      "animal",
      "bird",
      "bird of prey",
      "eagle"
      . . .
```

Scenario: Embedded images in PDFs

When the images you want to process are embedded in other files, such as PDF or DOCX, the enrichment pipeline will extract just the images and then pass them to OCR or image analysis for processing. Separation of image from text content occurs during the document cracking phase, and once the images are separated, they remain separate unless you explicitly merge the processed output back into the source text.

Text Merge is used to put image processing output back into the document. Although Text Merge is not a hard requirement, it's frequently invoked so that image output (OCR text, OCR layoutText, image tags, image captions) can be reintroduced into the document. Depending on the skill, the image output replaces an embedded binary image with an in-place text equivalent. Image Analysis output can be merged at image location. OCR output always appears at the end of each page.

The following workflow outlines the process of image extraction, analysis, merging, and how to extend the pipeline to push image-processed output into other text-based skills such as Entity Recognition or Text Translation.

1. After connecting to the data source, the indexer loads and cracks source documents, extracting images and text, and queuing each content type for processing. An enriched document consisting only of a root node (`"document"`) is created.

2. Images in the queue are normalized and passed into enriched documents as a "document/normalized_images" node.

3. Image enrichments execute, using `"/document/normalized_images"` as input.

4. Image outputs are passed into enriched documents, with each output as a separate node. Outputs vary by skill (text and layoutText for OCR, tags and captions for Image Analysis).

5. Optional but recommended if you want search documents to include both text and image-origin text together, Text Merge runs, combining the text representation of those images with the raw text extracted from the file. Text chunks are consolidated into a single large string, where the text is inserted first in the string and then the OCR text output or image tags and captions.

   The output of Text Merge is now the definitive text to analyze for any downstream skills that perform text processing. For example, if your skillset includes both OCR and Entity Recognition, the input to Entity Recognition should be `"document/merged_text"` (the targetName of the Text Merge skill output).

6. After all skills have executed, the enriched document is complete. In the last step, indexers refer to output field mappings to send enriched content to individual fields in the search index.

The following example skillset creates a `"merged_text"` field containing the original text of your document with embedded OCRed text in place of embedded images. It also includes an Entity Recognition skill that uses `"merged_text"` as input.

Request body syntax

JSON

```
{
  "description": "Extract text from images and merge with content text to
produce merged_text",
  "skills":
  [
    {
        "description": "Extract text (plain and structured) from image.",
        "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
        "context": "/document/normalized_images/*",
        "defaultLanguageCode": "en",
        "detectOrientation": true,
        "inputs": [
          {
            "name": "image",
```

```
            "source": "/document/normalized_images/*"
          }
        ],
        "outputs": [
          {
            "name": "text"
          }
        ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
      "description": "Create merged_text, which includes all the textual
representation of each image inserted at the right location in the content
field.",
      "context": "/document",
      "insertPreTag": " ",
      "insertPostTag": " ",
      "inputs": [
        {
          "name":"text", "source": "/document/content"
        },
        {
          "name": "itemsToInsert", "source":
"/document/normalized_images/*/text"
        },
        {
          "name":"offsets", "source":
"/document/normalized_images/*/contentOffset"
        }
      ],
      "outputs": [
        {
          "name": "mergedText", "targetName" : "merged_text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "context": "/document",
      "categories": [ "Person"],
      "defaultLanguageCode": "en",
      "minimumPrecision": 0.5,
      "inputs": [
        {
            "name": "text", "source": "/document/merged_text"
        }
      ],
      "outputs": [
        {
            "name": "persons", "targetName": "people"
        }
      ]
    }
  ]
}
```

Now that you have a merged_text field, you could map it as a searchable field in your indexer definition. All of the content of your files, including the text of the images, will be searchable.

Scenario: Visualize bounding boxes

Another common scenario is visualizing search results layout information. For example, you might want to highlight where a piece of text was found in an image as part of your search results.

Since the OCR step is performed on the normalized images, the layout coordinates are in the normalized image space. When displaying the normalized image, the presence of coordinates is generally not a problem, but in some situations you might want to display the original image. In this case, convert each of coordinate points in the layout to the original image coordinate system.

As a helper, if you need to transform normalized coordinates to the original coordinate space, you could use the following algorithm:

C#

```
/// <summary>
///  Converts a point in the normalized coordinate space to the original
coordinate space.
///  This method assumes the rotation angles are multiples of 90 degrees.
/// </summary>
public static Point GetOriginalCoordinates(Point normalized,
                            int originalWidth,
                            int originalHeight,
                            int width,
                            int height,
                            double rotationFromOriginal)
{
    Point original = new Point();
    double angle = rotationFromOriginal % 360;

    if (angle == 0 )
    {
        original.X = normalized.X;
        original.Y = normalized.Y;
    } else if (angle == 90)
    {
        original.X = normalized.Y;
        original.Y = (width - normalized.X);
    } else if (angle == 180)
    {
        original.X = (width -  normalized.X);
```

```
        original.Y = (height - normalized.Y);
    } else if (angle == 270)
    {
        original.X = height - normalized.Y;
        original.Y = normalized.X;
    }

    double scalingFactor = (angle % 180 == 0) ? originalHeight / height :
originalHeight / width;
    original.X = (int) (original.X * scalingFactor);
    original.Y = (int)(original.Y * scalingFactor);

    return original;
}
```

Scenario: Custom image skills

Images can also be passed into and returned from custom skills. A skillset base64-encodes the image being passed into the custom skill. To use the image within the custom skill, set `"/document/normalized_images/*/data"` as the input to the custom skill. Within your custom skill code, base64-decode the string before converting it to an image. To return an image to the skillset, base64-encode the image before returning it to the skillset.

The image is returned as an object with the following properties.

JSON

```
{
 "$type": "file",
 "data": "base64String"
}
```

The Azure Search Python samples repository has a complete sample implemented in Python of a custom skill that enriches images.

Passing images to custom skills

For scenarios where you require a custom skill to work on images, you can pass images to the custom skill, and have it return text or images. The Python sample image-processing demonstrates the workflow. The following skillset is from the sample.

The following skillset takes the normalized image (obtained during document cracking), and outputs slices of the image.

Sample skillset

JSON

```
{
  "description": "Extract text from images and merge with content text to
produce merged_text",
  "skills":
  [
    {
          "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
          "name": "ImageSkill",
          "description": "Segment Images",
          "context": "/document/normalized_images/*",
          "uri": "https://your.custom.skill.url",
          "httpMethod": "POST",
          "timeout": "PT30S",
          "batchSize": 100,
          "degreeOfParallelism": 1,
          "inputs": [
            {
              "name": "image",
              "source": "/document/normalized_images/*"
            }
          ],
          "outputs": [
            {
              "name": "slices",
              "targetName": "slices"
            }
          ],
          "httpHeaders": {}
        }
  ]
}
```

Custom skill example

The custom skill itself is external to the skillset. In this case, it is Python code that first loops through the batch of request records in the custom skill format, then converts the base64-encoded string to an image.

Python

```
# deserialize the request, for each item in the batch
for value in values:
  data = value['data']
  base64String = data["image"]["data"]
  base64Bytes = base64String.encode('utf-8')
  inputBytes = base64.b64decode(base64Bytes)
```

```
  # Use numpy to convert the string to an image
  jpg_as_np = np.frombuffer(inputBytes, dtype=np.uint8)
  # you now have an image to work with
```

Similarly to return an image, return a base64 encoded string within a JSON object with a `$type` property of `file`.

Python

```python
def base64EncodeImage(image):
    is_success, im_buf_arr = cv2.imencode(".jpg", image)
    byte_im = im_buf_arr.tobytes()
    base64Bytes = base64.b64encode(byte_im)
    base64String = base64Bytes.decode('utf-8')
    return base64String

 base64String = base64EncodeImage(jpg_as_np)
 result = {
  "$type": "file",
  "data": base64String
}
```

See also

- Create indexer (REST)
- Image Analysis skill
- OCR skill
- Text merge skill
- How to define a skillset
- How to map enriched fields
- How to pass images to custom skills

Enable caching for incremental enrichment in Azure Cognitive Search

Article • 04/24/2023

> **Important**
>
> This feature is in public preview under **supplemental terms of use**. The **preview REST API** supports this feature

This article explains how to add caching to an enrichment pipeline so that you can modify downstream enrichment steps without having to rebuild in full every time. By default, a skillset is stateless, and changing any part of its composition requires a full rerun of the indexer. With an **enrichment cache**, the indexer can determine which parts of the document tree must be refreshed based on changes detected in the skillset or indexer definitions. Existing processed output is preserved and reused wherever possible.

Cached content is placed in Azure Storage using account information that you provide. The container, named `ms-az-search-indexercache-<alpha-numerc-string>`, is created when you run the indexer. It should be considered an internal component managed by your search service and must not be modified.

If you're not familiar with setting up indexers, start with indexer overview and then continue on to skillsets to learn about enrichment pipelines. For more background on key concepts, see incremental enrichment.

Prerequisites

Azure Storage is used to store cached enrichments. The storage account must be general purpose v2.

Preview APIs or beta Azure SDKs are required for enabling cache on an indexer. The portal does not currently provide an option for caching enrichment.

> **Caution**
>
> If you're using the **SharePoint Online indexer (Preview)**, you should avoid incremental enrichment. Under certain circumstances, the cache becomes invalid, requiring an **indexer reset and run**, should you choose to reload it.

Enable on new indexers

On new indexers, add the "cache" property in the indexer definition payload when calling Create or Update Indexer (2021-04-30-Preview). You can also use the previous preview API version, 2020-06-30-Preview.

https

```
POST https://[service name].search.windows.net/indexers?api-version=2021-04-30-Preview
    {
        "name": "<YOUR-INDEXER-NAME>",
        "targetIndexName": "<YOUR-INDEX-NAME>",
        "dataSourceName": "<YOUR-DATASOURCE-NAME>",
        "skillsetName": "<YOUR-SKILLSET-NAME>",
        "cache" : {
            "storageConnectionString" : "<YOUR-STORAGE-ACCOUNT-CONNECTION-STRING>",
            "enableReprocessing": true
        },
        "fieldMappings" : [],
        "outputFieldMappings": [],
        "parameters": []
        }
    }
```

Enable on existing indexers

For existing indexers that already have a skillset, use the following steps to add caching. As a one-time operation, reset and rerun the indexer in full to load the cache.

Step 1: Get the indexer definition

Start with a valid, work indexer that has these components: data source, skillset, index. Using an API client, send a GET Indexer request to retrieve the indexer. When you use the preview API version to the GET the indexer, a "cache" property set to null is added to the definition automatically.

HTTP

```
GET https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?api-version=2021-04-30-Preview
    Content-Type: application/json
    api-key: [YOUR-ADMIN-KEY]
```

Step 2: Set the cache property

In the index definition, modify "cache" to include the following required and optional properties:

- (Required) `storageConnectionString` must be set to an Azure Storage connection string.
- (Optional) `enableReprocessing` boolean property (`true` by default), indicates that incremental enrichment is enabled. Set to `false` if you want to suspend incremental processing while other resource-intensive operations, such as indexing new documents, are underway and then switch back to `true` later.

HTTP

```
POST https://[service name].search.windows.net/indexers?api-version=2021-04-30-Preview
    {
        "name": "<YOUR-INDEXER-NAME>",
        "targetIndexName": "<YOUR-INDEX-NAME>",
        "dataSourceName": "<YOUR-DATASOURCE-NAME>",
        "skillsetName": "<YOUR-SKILLSET-NAME>",
        "cache" : {
            "storageConnectionString" : "<YOUR-STORAGE-ACCOUNT-CONNECTION-STRING>",
            "enableReprocessing": true
        },
        "fieldMappings" : [],
        "outputFieldMappings": [],
        "parameters": []
    }
```

Step 3: Reset the indexer

Reset Indexer is required when setting up incremental enrichment for existing indexers to ensure all documents are in a consistent state. You can use the portal or an API client for this task.

https

```
POST https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]/reset?api-version=2021-04-30-Preview
    Content-Type: application/json
    api-key: [YOUR-ADMIN-KEY]
```

Step 4: Save the indexer

Update Indexer (2021-04-30-Preview) with a PUT request, where the body of the request includes "cache".

```
HTTP

PUT https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?api-version=2021-04-30-Preview
    Content-Type: application/json
    api-key: [YOUR-ADMIN-KEY]
    {
        "name" : "<YOUR-INDEXER-NAME>",
        ...
        "cache": {
            "storageConnectionString": "<YOUR-STORAGE-ACCOUNT-CONNECTION-STRING>",
            "enableReprocessing": true
        }
    }
```

If you now issue another GET request on the indexer, the response from the service will include an `ID` property in the cache object. The alphanumeric string is appended to the name of the container containing all the cached results and intermediate state of each document processed by this indexer. The ID will be used to uniquely name the cache in Blob storage.

```
HTTP

    "cache": {
        "ID": "<ALPHA-NUMERIC STRING>",
        "enableReprocessing": true,
        "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<YOUR-STORAGE-ACCOUNT>;AccountKey=<YOUR-STORAGE-KEY>;EndpointSuffix=core.windows.net"
    }
```

Step 5: Run the indexer

To run indexer, you can use the portal or the API. In the portal, from the indexers list, select the indexer and click **Run**. One advantage to using the portal is that you can monitor indexer status, note the duration of the job, and how many documents are processed. Portal pages are refreshed every few minutes.

Alternatively, you can use REST to run the indexer:

```
HTTP
```

```
POST https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]/run?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [YOUR-ADMIN-KEY]
```

> **Note**
>
> A reset and rerun of the indexer results in a full rebuild so that content can be cached. All cognitive enrichments will be rerun on all documents. Reusing enriched content from the cache begins after the cache is loaded.

Check for cached output

Find the cache in Azure Storage, under Blob container. The container name will be `ms-az-search-indexercache-<some-alphanumeric-string>`.

A cache is created and used by an indexer. Its content is not human readable.

To verify whether the cache is operational, modify a skillset and run the indexer, then compare before-and-after metrics for execution time and document counts.

Skillsets that include image analysis and Optical Character Recognition (OCR) of scanned documents make good test cases. If you modify a downstream text skill or any skill that is not image-related, the indexer can retrieve all of the previously processed image and OCR content from cache, updating and processing only the text-related changes indicated by your edits. You can expect to see fewer documents in the indexer execution document count, shorter execution times, and fewer charges on your bill.

The file set used in cog-search-demo tutorials is a useful test case because it contains 14 files of various formats JPG, PNG, HTML, DOCX, PPTX, and other types. Change `en` to `es` or another language in the text translation skill for proof-of-concept testing of incremental enrichment.

Common errors

The following error occurs if you forget to specify a preview API version on the request:

`"The request is invalid. Details: indexer : A resource without a type name was found, but no expected type was specified. To allow entries without type information, the expected type must also be specified when the model is specified."`

A 400 Bad Request error will also occur if you are missing an indexer requirement. The error message will specify any missing dependencies.

Next steps

Incremental enrichment is applicable on indexers that contain skillsets, providing reusable content for both indexes and knowledge stores. The following links provide more information about caching and skillsets.

- Incremental enrichment (lifecycle and management)
- Skillset concepts and composition
- Create a skillset
- Tutorial: Use REST and AI to generate searchable content from Azure blobs

Tips for AI enrichment in Azure Cognitive Search

Article • 09/16/2022 • 5 minutes to read

This article contains a list of tips and tricks to keep you moving as you get started with AI enrichment capabilities in Azure Cognitive Search.

If you haven't already, step through Quickstart: Create a skillset for AI enrichment for a light-weight introduction to enrichment of blob data.

Tip 1: Start with a small dataset

The best way to find issues quickly is to increase the speed at which you can fix issues, which means working with smaller or simpler documents.

Start by creating a data source with just a handful of documents or rows in a table that are representative of the documents that will be indexed.

Run your sample through the end-to-end pipeline and check that the results meet your needs. Once you're satisfied with the results, you're ready to add more files to your data source.

Tip 2: Make sure your data source credentials are correct

The data source connection isn't validated until you define an indexer that uses it. If you get connection errors, make sure that:

- Your connection string is correct. Specially when you're creating SAS tokens, make sure to use the format expected by Azure Cognitive Search. See How to specify credentials section to learn about the different formats supported.
- Your container name in the indexer is correct.

Tip 3: See what works even if there are some failures

Sometimes a small failure stops an indexer in its tracks. That is fine if you plan to fix issues one by one. However, you might want to ignore a particular type of error,

allowing the indexer to continue so that you can see what flows are actually working.

In that case, you may want to tell the indexer to ignore errors. Do that by setting *maxFailedItems* and *maxFailedItemsPerBatch* as -1 as part of the indexer definition.

```
{
  "// rest of your indexer definition
   "parameters":
   {
      "maxFailedItems":-1,
      "maxFailedItemsPerBatch":-1
   }
}
```

> **Note**
>
> As a best practice, set the maxFailedItems, maxFailedItemsPerBatch to 0 for production workloads

Tip 4: Use Debug sessions to identify and resolve issues with your skillset

Debug sessions is a visual editor that works with an existing skillset in the Azure portal. Within a debug session you can identify and resolve errors, validate changes, and commit changes to a production skillset in the AI enrichment pipeline. This is a preview feature read the documentation. For more information about concepts and getting started, see Debug sessions.

Debug sessions work on a single document are a great way for you to iteratively build more complex enrichment pipelines.

Tip 5: Looking at enriched documents under the hood

Enriched documents are temporary structures created during enrichment, and then deleted when processing is complete.

To capture a snapshot of the enriched document created during indexing, add a field called `enriched` to your index. The indexer automatically dumps into the field a string

representation of all the enrichments for that document.

The `enriched` field will contain a string that is a logical representation of the in-memory enriched document in JSON. The field value is a valid JSON document, however. Quotes are escaped so you'll need to replace `\"` with `"` in order to view the document as formatted JSON.

The enriched field is intended for debugging purposes only, to help you understand the logical shape of the content that expressions are being evaluated against. You shouldn't depend on this field for indexing purposes.

Add an `enriched` field as part of your index definition for debugging purposes:

Request Body Syntax

JSON

```
{
  "fields": [
    // other fields go here.
    {
      "name": "enriched",
      "type": "Edm.String",
      "searchable": false,
      "sortable": false,
      "filterable": false,
      "facetable": false
    }
  ]
}
```

Tip 6: Expected content fails to appear

Missing content could be the result of documents getting dropped during indexing. Free and Basic tiers have low limits on document size. Any file exceeding the limit is dropped during indexing. You can check for dropped documents in the Azure portal. In the search service dashboard, double-click the Indexers tile. Review the ratio of successful documents indexed. If it isn't 100%, you can select the ratio to get more detail.

If the problem is related to file size, you might see an error like this: "The blob <file-name>" has the size of <file-size> bytes, which exceed the maximum size for document extraction for your current service tier." For more information on indexer limits, see Service limits.

A second reason for content failing to appear might be related input/output mapping errors. For example, an output target name is "People" but the index field name is lower-case "people". The system could return 201 success messages for the entire pipeline so you think indexing succeeded, when in fact a field is empty.

Tip 7: Extend processing beyond maximum run time (24-hour window)

Image analysis is computationally intensive for even simple cases, so when images are especially large or complex, processing times can exceed the maximum time allowed.

Maximum run time varies by tier: several minutes on the Free tier, 24-hour indexing on billable tiers. If processing fails to complete within a 24-hour period for on-demand processing, switch to a schedule to have the indexer pick up processing where it left off.

For scheduled indexers, indexing resumes on schedule at the last known good document. By using a recurring schedule, the indexer can work its way through the image backlog over a series of hours or days, until all unprocessed images are processed. For more information on schedule syntax, see Schedule an indexer.

> **Note**
>
> If an indexer is set to a certain schedule but repeatedly fails on the same document over and over again each time it runs, the indexer will begin running on a less frequent interval (up to the maximum of at least once every 24 hours) until it successfully makes progress again. If you believe you have fixed whatever the issue that was causing the indexer to be stuck at a certain point, you can perform an on-demand run of the indexer, and if that successfully makes progress, the indexer will return to its set schedule interval again.

For portal-based indexing (as described in the quickstart), choosing the "run once" indexer option limits processing to 1 hour (`"maxRunTime": "PT1H"`). You might want to extend the processing window to something longer.

Tip 8: Increase indexing throughput

For parallel indexing, place your data into multiple containers or multiple virtual folders inside the same container. Then create multiple data source and indexer pairs. All indexers can use the same skillset and write into the same target search index, so your search app doesn't need to be aware of this partitioning.

See also

- Quickstart: Create an AI enrichment pipeline in the portal
- Tutorial: Learn AI enrichment REST APIs
- How to configure blob indexers
- How to define a skillset
- How to map enriched fields to an index

Add a custom skill to an Azure Cognitive Search enrichment pipeline

Article • 06/21/2022 • 5 minutes to read

An enrichment pipeline can include both built-in skills and custom skills that you personally create and publish. Your custom code executes externally to the search service (for example, as an Azure function), but accepts inputs and sends outputs to the skillset just like any other skill.

Custom skills might sound complex but can be simple and straightforward in terms of implementation. If you have existing packages that provide pattern matching or classification models, the content you extract from blobs could be passed to these models for processing. Since AI enrichment is Azure-based, your model should be on Azure also. Some common hosting methodologies include using Azure Functions or Containers.

If you are building a custom skill, this article describes the interface you'll use to integrate the skill into the pipeline. The primary requirement is the ability to accept inputs and emit outputs in ways that are consumable within the skillset as a whole. As such, the focus of this article is on the input and output formats that the enrichment pipeline requires.

Benefits of custom skills

Building a custom skill gives you a way to insert transformations unique to your content. A custom skill executes independently, applying whatever enrichment step you require. For example, you could build custom classification models to differentiate business and financial contracts and documents, or add a speech recognition skill to reach deeper into audio files for relevant content. For a step-by-step example, see Example: Creating a custom skill for AI enrichment.

Set the endpoint and timeout interval

The interface for a custom skill is specified through the Custom Web API skill.

JSON

```
"@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
"description": "This skill has a 230 second timeout",
"uri": "https://[your custom skill uri goes here]",
```

```
"authResourceId": "[for managed identity connections, your app's client ID
goes here]",
"timeout": "PT230S",
```

The URI is the HTTPS endpoint of your function or app. When setting the URI, make sure the URI is secure (HTTPS). If your code is hosted in an Azure function app, the URI should include an API key in the header or as a URI parameter to authorize the request.

If instead your function or app uses Azure managed identities and Azure roles for authentication and authorization, the custom skill can include an authentication token on the request. The following points describe the requirements for this approach:

- The search service, which sends the request on the indexer's behalf, must be configured to use a managed identity (either system or user-assigned) so that the caller can be authenticated by Azure Active Directory.
- Your function or app must be configured for Azure Active Directory.
- Your custom skill definition must include an "authResourceId" property. This property takes an application (client) ID, in a supported format: `api://<appId>`.

By default, the connection to the endpoint will time out if a response is not returned within a 30-second window. The indexing pipeline is synchronous and indexing will produce a timeout error if a response is not received in that time frame. You can increase the interval to a maximum value of 230 seconds by setting the timeout parameter:

Format Web API inputs

The Web API must accept an array of records to be processed. Each record must contain a property bag that is the input provided to your Web API.

Suppose you want to create a simple enricher that identifies the first date mentioned in the text of a contract. In this example, the custom skill accepts a single input "contractText" as the contract text. The skill also has a single output, which is the date of the contract. To make the enricher more interesting, return this "contractDate" in the shape of a multi-part complex type.

Your Web API should be ready to receive a batch of input records. Each member of the "values" array represents the input for a particular record. Each record is required to have the following elements:

- A "recordId" member that is the unique identifier for a particular record. When your enricher returns the results, it must provide this "recordId" in order to allow

the caller to match the record results to their input.

- A "data" member, which is essentially a bag of input fields for each record.

The resulting Web API request might look like this:

JSON

```
{
    "values": [
      {
        "recordId": "a1",
        "data":
           {
             "contractText":
                "This is a contract that was issues on November 3, 2017 and
that involves... "
           }
      },
      {
        "recordId": "b5",
        "data":
           {
             "contractText":
                "In the City of Seattle, WA on February 5, 2018 there was a
decision made..."
           }
      },
      {
        "recordId": "c3",
        "data":
           {
             "contractText": null
           }
      }
    ]
}
```

In practice, your code may get called with hundreds or thousands of records instead of only the three shown here.

Format Web API outputs

The format of the output is a set of records containing a "recordId", and a property bag. This particular example has only one output, but you could output more than one property. As a best practice, consider returning error and warning messages if a record could not be processed.

JSON

```
{
  "values":
  [
      {
        "recordId": "b5",
        "data" :
        {
            "contractDate":  { "day" : 5, "month": 2, "year" : 2018 }
        }
      },
      {
        "recordId": "a1",
        "data" : {
            "contractDate": { "day" : 3, "month": 11, "year" : 2017 }
        }
      },
      {
        "recordId": "c3",
        "data" :
        {
        },
        "errors": [ { "message": "contractText field required "}   ],
        "warnings": [ {"message": "Date not found" }  ]
      }
    ]
}
```

Add a custom skill to a skillset

When you create a Web API enricher, you can describe HTTP headers and parameters as part of the request. The snippet below shows how request parameters and optional HTTP headers can be included in the skillset definition. Setting an HTTP header is useful if you need to pass configuration settings to your code.

JSON

```
{
    "skills": [
      {
        "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
        "name": "myCustomSkill",
        "description": "This skill calls an Azure function, which in turn
calls TA sentiment",
        "uri": "https://indexer-e2e-
webskill.azurewebsites.net/api/DateExtractor?language=en",
        "context": "/document",
        "httpHeaders": {
            "DateExtractor-Api-Key": "foo"
        },
        "inputs": [
```

```
          {
            "name": "contractText",
            "source": "/document/content"
          }
        ],
        "outputs": [
          {
            "name": "contractDate",
            "targetName": "date"
          }
        ]
      }
  ]
}
```

Watch this video

For a video introduction and demo, watch the following demo.
https://www.youtube-nocookie.com/embed/fHLCE-NZeb4?version=3

Next steps

This article covered the interface requirements necessary for integrating a custom skill into a skillset. Continue with these links to learn more about custom skills and skillset composition.

- Power Skills: a repository of custom skills
- Example: Creating a custom skill for AI enrichment
- How to define a skillset
- Create Skillset (REST)
- How to map enriched fields

Efficiently scale out a custom skill

Article • 12/02/2022 • 4 minutes to read

Custom skills are web APIs that implement a specific interface. A custom skill can be implemented on any publicly addressable resource. The most common implementations for custom skills are:

- Azure Functions for custom logic skills
- Azure Webapps for simple containerized AI skills
- Azure Kubernetes service for more complex or larger skills.

Prerequisites

- Review the custom skill interface for an introduction into the input/output interface that a custom skill should implement.

- Set up your environment. You could start with this tutorial end-to-end to set up serverless Azure Function using Visual Studio Code and Python extensions.

Skillset configuration

Configuring a custom skill for maximizing throughput of the indexing process requires an understanding of the skill, indexer configurations and how the skill relates to each document. For example, the number of times a skill is invoked per document and the expected duration per invocation.

Skill settings

On the custom skill set the following parameters.

1. Set `batchSize` of the custom skill to configure the number of records sent to the skill in a single invocation of the skill.

2. Set the `degreeOfParallelism` to calibrate the number of concurrent requests the indexer will make to your skill.

3. Set `timeout` to a value sufficient for the skill to respond with a valid response.

4. In the `indexer` definition, set batchSize to the number of documents that should be read from the data source and enriched concurrently.

Considerations

Setting these variables to optimize the indexers performance requires determining if your skill performs better with many concurrent small requests or fewer large requests. A few questions to consider are:

- What is the skill invocation cardinality? Does the skill execute once for each document, for instance a document classification skill, or could the skill execute multiple times per document, a paragraph classification skill?
- On average how many documents are read from the data source to fill out a skill request based on the skill batch size? Ideally, this should be less than the indexer batch size. With batch sizes greater than 1 your skill can receive records from multiple source documents. For example if the indexer batch count is 5 and the skill batch count is 50 and each document generates only five records, the indexer will need to fill a custom skill request across multiple indexer batches.
- The average number of requests an indexer batch can generate should give you an optimal setting for the degrees of parallelism. If your infrastructure hosting the skill cannot support that level of concurrency, consider dialing down the degrees of parallelism. As a best practice, test your configuration with a few documents to validate your choices on the parameters.
- Testing with a smaller sample of documents, evaluate the execution time of your skill to the overall time taken to process the subset of documents. Does your indexer spend more time building a batch or waiting for a response from your skill?
- Consider the upstream implications of parallelism. If the input to a custom skill is an output from a prior skill, are all the skills in the skillset scaled out effectively to minimize latency?

Error handling in the custom skill

Custom skills should return a success status code HTTP 200 when the skill completes successfully. If one or more records in a batch result in errors, consider returning multi-status code 207. The errors or warnings list for the record should contain the appropriate message.

Any items in a batch that errors will result in the corresponding document failing. If you need the document to succeed, return a warning.

Any status code over 299 is evaluated as an error and all the enrichments are failed resulting in a failed document.

Common error messages

- `Could not execute skill because it did not execute within the time limit '00:00:30'. This is likely transient. Please try again later. For custom skills, consider increasing the 'timeout' parameter on your skill in the skillset.` Set the timeout parameter on the skill to allow for a longer execution duration.
- `Could not execute skill because Web Api skill response is invalid.` Indicative of the skill not returning a message in the custom skill response format. This could be the result of an uncaught exception in the skill.
- `Could not execute skill because the Web Api request failed.` Most likely caused by authorization errors or exceptions.
- `Could not execute skill.` Commonly the result of the skill response being mapped to an existing property in the document hierarchy.

Testing custom skills

Start by testing your custom skill with a REST API client to validate:

- The skill implements the custom skill interface for requests and responses
- The skill returns valid JSON with the `application/JSON` MIME type
- Returns a valid HTTP status code

Create a debug session to add your skill to the skillset and make sure it produces a valid enrichment. While a debug session does not allow you to tune the performance of the skill, it enables you to ensure that the skill is configured with valid values and returns the expected enriched objects.

Best practices

- While skills can accept and return larger payloads, consider limiting the response to 150 MB or less when returning JSON.

- Consider setting the batch size on the indexer and skill to ensure that each data source batch generates a full payload for your skill.
- For long running tasks, set the timeout to a high enough value to ensure the indexer does not error out when processing documents concurrently.
- Optimize the indexer batch size, skill batch size, and skill degrees of parallelism to generate the load pattern your skill expects, fewer large requests or many small requests.
- Monitor custom skills with detailed logs of failures as you can have scenarios where specific requests consistently fail as a result of the data variability.

Next steps

Congratulations! Your custom skill is now scaled right to maximize throughput on the indexer.

- Power Skills: a repository of custom skills
- Add a custom skill to an AI enrichment pipeline
- Add a Azure Machine Learning skill
- Use debug sessions to test changes

Example: Create a custom skill using the Bing Entity Search API

Article • 12/02/2022 • 8 minutes to read

In this example, learn how to create a web API custom skill. This skill will accept locations, public figures, and organizations, and return descriptions for them. The example uses an Azure Function to wrap the Bing Entity Search API so that it implements the custom skill interface.

Prerequisites

- Read about custom skill interface article if you aren't familiar with the input/output interface that a custom skill should implement.
- Create a Bing Search v7 resource through the Azure Portal. A free tier is available and sufficient for this example.
- Install Visual Studio 2019 or later, including the Azure development workload.

Create an Azure Function

Although this example uses an Azure Function to host a web API, it isn't required. As long as you meet the interface requirements for a cognitive skill, the approach you take is immaterial. Azure Functions, however, make it easy to create a custom skill.

Create a function app

1. In Visual Studio, select **New** > **Project** from the File menu.
2. In the New Project dialog, select **Azure Functions** as the template and select **Next**. Type a name for your project, and select **Create**. The function app name must be valid as a C# namespace, so don't use underscores, hyphens, or any other non-alphanumeric characters.
3. Select the type to be **HTTP Trigger**
4. For Storage Account, you may select **None**, as you won't need any storage for this function.
5. Select **Create** to create the function project and HTTP triggered function.

Modify the code to call the Bing Entity Search Service

Visual Studio creates a project and in it a class that contains boilerplate code for the chosen function type. The *FunctionName* attribute on the method sets the name of the function. The *HttpTrigger* attribute specifies that the function is triggered by an HTTP request.

Now, replace all of the content of the file *Function1.cs* with the following code:

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;

namespace SampleSkills
{
    /// <summary>
    /// Sample custom skill that wraps the Bing entity search API to connect
it with a
    /// AI enrichment pipeline.
    /// </summary>
    public static class BingEntitySearch
    {
        #region Credentials
        // IMPORTANT: Make sure to enter your credential and to verify the
API endpoint matches yours.
        static readonly string bingApiEndpoint =
"https://api.bing.microsoft.com/v7.0/entities";
        static readonly string key = "<enter your api key here>";
        #endregion

        #region Class used to deserialize the request
        private class InputRecord
        {
            public class InputRecordData
            {
                public string Name { get; set; }
            }

            public string RecordId { get; set; }
            public InputRecordData Data { get; set; }
        }
```

```
        private class WebApiRequest
        {
            public List<InputRecord> Values { get; set; }
        }
        #endregion

        #region Classes used to serialize the response

        private class OutputRecord
        {
            public class OutputRecordData
            {
                public string Name { get; set; } = "";
                public string Description { get; set; } = "";
                public string Source { get; set; } = "";
                public string SourceUrl { get; set; } = "";
                public string LicenseAttribution { get; set; } = "";
                public string LicenseUrl { get; set; } = "";
            }

            public class OutputRecordMessage
            {
                public string Message { get; set; }
            }

            public string RecordId { get; set; }
            public OutputRecordData Data { get; set; }
            public List<OutputRecordMessage> Errors { get; set; }
            public List<OutputRecordMessage> Warnings { get; set; }
        }

        private class WebApiResponse
        {
            public List<OutputRecord> Values { get; set; }
        }
        #endregion

        #region Classes used to interact with the Bing API
        private class BingResponse
        {
            public BingEntities Entities { get; set; }
        }
        private class BingEntities
        {
            public BingEntity[] Value { get; set; }
        }

        private class BingEntity
        {
            public class EntityPresentationinfo
            {
                public string[] EntityTypeHints { get; set; }
            }
```

```
            public class License
            {
                public string Url { get; set; }
            }

            public class ContractualRule
            {
                public string _type { get; set; }
                public License License { get; set; }
                public string LicenseNotice { get; set; }
                public string Text { get; set; }
                public string Url { get; set; }
            }

            public ContractualRule[] ContractualRules { get; set; }
            public string Description { get; set; }
            public string Name { get; set; }
            public EntityPresentationinfo EntityPresentationInfo { get; set;
}
        }
        #endregion

        #region The Azure Function definition

        [FunctionName("EntitySearch")]
        public static async Task<IActionResult> Run(
            [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)]
HttpRequest req,
            ILogger log)
        {
            log.LogInformation("Entity Search function: C# HTTP trigger
function processed a request.");

            var response = new WebApiResponse
            {
                Values = new List<OutputRecord>()
            };

            string requestBody = new StreamReader(req.Body).ReadToEnd();
            var data = JsonConvert.DeserializeObject<WebApiRequest>
(requestBody);

            // Do some schema validation
            if (data == null)
            {
                return new BadRequestObjectResult("The request schema does
not match expected schema.");
            }
            if (data.Values == null)
            {
                return new BadRequestObjectResult("The request schema does
not match expected schema. Could not find values array.");
            }

            // Calculate the response for each value.
```

```
            foreach (var record in data.Values)
            {
                if (record == null || record.RecordId == null) continue;

                OutputRecord responseRecord = new OutputRecord
                {
                    RecordId = record.RecordId
                };

                try
                {
                    responseRecord.Data =
GetEntityMetadata(record.Data.Name).Result;
                }
                catch (Exception e)
                {
                    // Something bad happened, log the issue.
                    var error = new OutputRecord.OutputRecordMessage
                    {
                        Message = e.Message
                    };

                    responseRecord.Errors = new
List<OutputRecord.OutputRecordMessage>
                    {
                        error
                    };
                }
                finally
                {
                    response.Values.Add(responseRecord);
                }
            }

            return (ActionResult)new OkObjectResult(response);
        }

        #endregion

        #region Methods to call the Bing API
        /// <summary>
        /// Gets metadata for a particular entity based on its name using
Bing Entity Search
        /// </summary>
        /// <param name="entityName">The name of the entity to extract data
for.</param>
        /// <returns>Asynchronous task that returns entity data. </returns>
        private async static Task<OutputRecord.OutputRecordData>
GetEntityMetadata(string entityName)
        {
            var uri = bingApiEndpoint + "?q=" + entityName + "&mkt=en-
us&count=10&offset=0&safesearch=Moderate";
            var result = new OutputRecord.OutputRecordData();

            using (var client = new HttpClient())
```

```
            using (var request = new HttpRequestMessage {
                Method = HttpMethod.Get,
                RequestUri = new Uri(uri)
            })
            {
                request.Headers.Add("Ocp-Apim-Subscription-Key", key);

                HttpResponseMessage response = await
client.SendAsync(request);
                string responseBody = await
response?.Content?.ReadAsStringAsync();

                BingResponse bingResult =
JsonConvert.DeserializeObject<BingResponse>(responseBody);
                if (bingResult != null)
                {
                    // In addition to the list of entities that could match
the name, for simplicity let's return information
                    // for the top match as additional metadata at the root
object.
                    return AddTopEntityMetadata(bingResult.Entities?.Value);
                }
            }

            return result;
        }

        private static OutputRecord.OutputRecordData
AddTopEntityMetadata(BingEntity[] entities)
        {
            if (entities != null)
            {
                foreach (BingEntity entity in entities.Where(
                    entity =>
entity?.EntityPresentationInfo?.EntityTypeHints != null
                        && (entity.EntityPresentationInfo.EntityTypeHints[0]
== "Person"
                            ||
entity.EntityPresentationInfo.EntityTypeHints[0] == "Organization"
                            ||
entity.EntityPresentationInfo.EntityTypeHints[0] == "Location")
                        && !String.IsNullOrEmpty(entity.Description)))
                {
                    var rootObject = new OutputRecord.OutputRecordData
                    {
                        Description = entity.Description,
                        Name = entity.Name
                    };

                    if (entity.ContractualRules != null)
                    {
                        foreach (var rule in entity.ContractualRules)
                        {
                            switch (rule._type)
                            {
```

```
                                case "ContractualRules/LicenseAttribution":
                                    rootObject.LicenseAttribution =
rule.LicenseNotice;
                                    rootObject.LicenseUrl =
rule.License.Url;
                                    break;
                                case "ContractualRules/LinkAttribution":
                                    rootObject.Source = rule.Text;
                                    rootObject.SourceUrl = rule.Url;
                                    break;
                            }
                        }
                    }

                    return rootObject;
                }
            }

            return new OutputRecord.OutputRecordData();
        }
        #endregion
    }
}
```

Make sure to enter your own *key* value in the `key` constant based on the key you got when signing up for the Bing entity search API.

This sample includes all necessary code in a single file for convenience. You can find a slightly more structured version of that same skill in the power skills repository.

Of course, you may rename the file from `Function1.cs` to `BingEntitySearch.cs`.

Test the function from Visual Studio

Press **F5** to run the program and test function behaviors. In this case, we'll use the function below to look up two entities. Use Postman or Fiddler to issue a call like the one shown below:

HTTP

```
POST https://localhost:7071/api/EntitySearch
```

Request body

JSON

```
{
    "values": [
        {
            "recordId": "e1",
            "data":
            {
                "name":  "Pablo Picasso"
            }
        },
        {
            "recordId": "e2",
            "data":
            {
                "name":  "Microsoft"
            }
        }
    ]
}
```

Response

You should see a response similar to the following example:

JSON

```
{
    "values": [
        {
            "recordId": "e1",
            "data": {
                "name": "Pablo Picasso",
                "description": "Pablo Ruiz Picasso was a Spanish painter
[...]",
                "source": "Wikipedia",
                "sourceUrl": "http://en.wikipedia.org/wiki/Pablo_Picasso",
                "licenseAttribution": "Text under CC-BY-SA license",
                "licenseUrl": "http://creativecommons.org/licenses/by-
sa/3.0/"
            },
            "errors": null,
            "warnings": null
        },
        "..."
    ]
}
```

Publish the function to Azure

When you're satisfied with the function behavior, you can publish it.

1. In **Solution Explorer**, right-click the project and select **Publish**. Choose **Create New > Publish**.

2. If you haven't already connected Visual Studio to your Azure account, select **Add an account....**

3. Follow the on-screen prompts. You're asked to specify a unique name for your app service, the Azure subscription, the resource group, the hosting plan, and the storage account you want to use. You can create a new resource group, a new hosting plan, and a storage account if you don't already have these. When finished, select **Create**

4. After the deployment is complete, notice the Site URL. It is the address of your function app in Azure.

5. In the Azure portal, navigate to the Resource Group, and look for the `EntitySearch` Function you published. Under the **Manage** section, you should see Host Keys. Select the **Copy** icon for the *default* host key.

Test the function in Azure

Now that you have the default host key, test your function as follows:

HTTP

```
POST https://[your-entity-search-app-name].azurewebsites.net/api/EntitySearch?code=[enter default host key here]
```

Request Body

JSON

```
{
    "values": [
        {
            "recordId": "e1",
            "data":
            {
                "name":  "Pablo Picasso"
            }
        },
        {
            "recordId": "e2",
```

```
            "data":
            {
                "name":  "Microsoft"
            }
        }
    ]
}
```

This example should produce the same result you saw previously when running the function in the local environment.

Connect to your pipeline

Now that you have a new custom skill, you can add it to your skillset. The example below shows you how to call the skill to add descriptions to organizations in the document (this could be extended to also work on locations and people). Replace `[your-entity-search-app-name]` with the name of your app.

JSON

```
{
    "skills": [
      "[... your existing skills remain here]",
      {
        "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
        "description": "Our new Bing entity search custom skill",
        "uri": "https://[your-entity-search-app-
name].azurewebsites.net/api/EntitySearch?code=[enter default host key
here]",
          "context": "/document/merged_content/organizations/*",
          "inputs": [
            {
              "name": "name",
              "source": "/document/merged_content/organizations/*"
            }
          ],
          "outputs": [
            {
              "name": "description",
              "targetName": "description"
            }
          ]
      }
  ]
}
```

Here, we're counting on the built-in entity recognition skill to be present in the skillset and to have enriched the document with the list of organizations. For reference, here's

an entity extraction skill configuration that would be sufficient in generating the data we need:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "name": "#1",
    "description": "Organization name extraction",
    "context": "/document/merged_content",
    "categories": [ "Organization" ],
    "defaultLanguageCode": "en",
    "inputs": [
        {
            "name": "text",
            "source": "/document/merged_content"
        },
        {
            "name": "languageCode",
            "source": "/document/language"
        }
    ],
    "outputs": [
        {
            "name": "organizations",
            "targetName": "organizations"
        }
    ]
},
```

Next steps

Congratulations! You've created your first custom skill. Now you can follow the same pattern to add your own custom functionality. Click the following links to learn more.

- Power Skills: a repository of custom skills
- Add a custom skill to an AI enrichment pipeline
- How to define a skillset
- Create Skillset (REST)
- How to map enriched fields

Create a knowledge store using REST and Postman

Article • 03/24/2023 • 14 minutes to read

Knowledge store is a feature of Azure Cognitive Search that accepts output from an AI enrichment pipeline and makes it available in Azure Storage for downstream apps and workloads. After the knowledge store is populated, use tools like Storage Explorer or Power BI to explore the content.

In this article, you'll learn how to use the REST API to ingest, enrich, and explore a set of customer reviews of hotel stays in a knowledge store. The knowledge store contains original text content pulled from the source, plus AI-generated content that includes a sentiment score, key phrase extraction, language detection, and text translation of non-English customer comments.

To make the initial data set available, the hotel reviews are first imported into Azure Blob Storage. Post-processing, the results are saved as a knowledge store in Azure Table Storage.

> **Note**
>
> This article provides detailed explanations of each step. For a faster approach, see **Create a knowledge store in Azure portal** instead.

Prerequisites

- Postman app
- An Azure account with an active subscription. Create an account for free.
- Azure Cognitive Search. Create a service or find an existing one. You can use the free service for this exercise.
- Azure Storage. Create an account or find an existing one. The account type must be **StorageV2 (general purpose V2)**.
- Sample data loaded into Blob Storage (instructions provided in the next section).

Load data

This step uses Azure Cognitive Search, Azure Blob Storage, and Azure Cognitive Services for the AI. Because the workload is so small, Cognitive Services is tapped behind the scenes to provide free processing for up to 20 transactions daily. A small workload means that you can skip creating or attaching a Cognitive Services resource.

1. Download HotelReviews_Free.csv. This CSV contains 19 pieces of customer feedback about a single hotel (originates from Kaggle.com). The file is in a repo with other sample data. If you don't want the whole repo, copy the raw content and paste it into a spreadsheet app on your device.

2. In Azure portal, on the Azure Storage resource page, use **Storage Browser** to create a blob container named **hotel-reviews**.

3. Select **Upload** at the top of the page to load the **HotelReviews-Free.csv** file you downloaded from the previous step.

Decide on a connection strategy

During skillset execution, the indexer connects to Azure Storage and creates the knowledge store. The connection information will be specified in the "knowledgeStore" section of the skillset. You can choose from the following approaches when setting up your connection:

- Option 1: Obtain a full access Azure Storage connection string that includes an access key:

  In the Azure Storage portal page, select **Access Keys** on the left navigation pane.

  In **Access Keys**, select **Show Keys** at the top of the page to unhide the connection strings, and then copy the connection string for either key1 or key2.

  A full access connection string has the following format:

```JSON
"knowledgeStore": {
    "storageConnectionString":
"DefaultEndpointsProtocol=https;AccountName=<YOUR-ACCOUNT-
NAME>;AccountKey=<YOUR-ACCOUNT-KEY>;EndpointSuffix=core.windows.net;"
}
```

- Option 2: Use your search service's system managed identity or user-assigned managed identity to connect to Azure Storage. Follow the instructions and examples in Connect using a managed identity. You'll need to set up the managed identity, assign roles, and assemble a connection string.

  A connection string for a system managed identity has the following format:

```JSON
"knowledgeStore": {
    "storageConnectionString": "ResourceId=/subscriptions/{YOUR-
SUBSCRIPTION-ID}/resourceGroups/{YOUR-RESOURCE-GROUP-
NAME}/providers/Microsoft.Storage/storageAccounts/{YOUR-ACCOUNT-NAME};"
}
```

Once you decide on an approach, you'll provide the connection string as a variable in the Postman collection, as described in the next step.

Configure requests

1. Download the azure-search-postman-samples from GitHub and unzip the file. There are multiple collections in the repo. You'll use the collection in the knowledge-store folder.

2. In Postman, select **File** > **Import** to import the KnowledgeStore.postman_collection.json file.

3. Select the **Collections** tab, and then select the **...** (ellipsis) button.

4. Select **Edit**.

5. In the **Edit** dialog box, select the **Variables** tab.

On the **Variables** tab, you can add values that Postman swaps in every time it encounters a specific variable inside double braces. For example, Postman replaces the symbol `{{admin-key}}` with the current value that you set for `admin-key`. Postman makes the substitution in URLs, headers, the request body, and so on.

Variables are defined for Azure services, service connections, and object names. Replace the service and connection placeholder values with actual values for your search service and storage account. You can find these values in the Azure portal.

- To get the values for `search-service-name` and `search-service-admin-key`, go to the Azure Cognitive Search service in the portal and copy the values from **Overview** and **Keys** pages.
- To get the values for `storage-account-name` and `storage-account-connection-string`, check the **Access Keys** page in the portal, or format a connection string that references a managed identity.

Variable	Where to get it
`admin-key`	On the **Keys** page of the Azure Cognitive Search service.
`api-version`	Leave as **2020-06-30**.
`datasource-name`	Leave as **hotel-reviews-ds**.
`indexer-name`	Leave as **hotel-reviews-ixr**.
`index-name`	Leave as **hotel-reviews-ix**.
`search-service-name`	The name of the Azure Cognitive Search service. If the URL is `https://mySearchService.search.windows.net`, the value you should enter is `mySearchService`.
`skillset-name`	Leave as **hotel-reviews-ss**.
`storage-account-name`	The Azure storage account name.

Variable	Where to get it
`storage-connection-string`	Use the storage account's connection string from **Access Keys** or paste in a connection string that references a managed identity.
`storage-container-name`	Leave as **hotel-reviews**.

Review the request collection in Postman

Knowledge stores are defined in skillsets, which are in turn attached to indexers. Creating a knowledge store requires that you create all of the upstream objects, including an index, data source, skillset, and indexer. Although an index is unrelated to a knowledge store, an indexer requires it for execution, so you will create one as an indexer prerequisite.

When you create a knowledge store, you'll issue four HTTP requests:

- **PUT request to create the index**: This index will contain searchable data returned in query requests issued to your search service.
- **POST request to create the datasource**: This data source provides connection information to the indexer. The indexer connects to your Azure Storage account to retrieve the sample data.
- **PUT request to create the skillset**: The skillset specifies the enrichments that are applied to your data. It also specifies the structure of the knowledge store.
- **PUT request to create the indexer**: Running the indexer reads the data, applies the skillset, creates the knowledge store in Azure Storage, and saves the results. You must run this request last.

The variables you set up earlier are used in the headers and URL. The following screenshot for the Create Index request shows where these variables appear in the request.

> **Note**
>
> All of the requests in the collection set `api-key` and `Content-type` headers, which are required. If Postman recognizes a variable, the variable appears in orange text, as with `{{admin-key}}` in the preceding screenshot. If the variable is misspelled, it appears in red text.

Create an index

Use the Create Index (REST API) to create a search index on the search service. A search index is unrelated to a knowledge store, but the indexer requires that you create one. The search index will contain the same content as the knowledge store. If you want an alternative approach for exploring your content, you can query this index by sending query requests to your search service.

Create the index by issuing a PUT request to `https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}?api-version={{api-version}}`. The index schema is provided in the body of the quest.

JSON

```
{
    "name": "{{index-name}}",
    "fields": [
        { "name": "name", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
        { "name": "reviews_date", "type": "Edm.DateTimeOffset",
"searchable": false, "filterable": false, "sortable": false, "facetable":
false },
        { "name": "reviews_rating", "type": "Edm.String", "searchable":
false, "filterable": false, "sortable": false, "facetable": false },
        { "name": "reviews_text", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
        { "name": "reviews_title", "type": "Edm.String", "searchable":
```

```
false, "filterable": false, "sortable": false, "facetable": false },
        { "name": "reviews_username", "type": "Edm.String", "searchable":
false, "filterable": false, "sortable": false, "facetable": false },
        { "name": "AzureSearch_DocumentKey", "type": "Edm.String",
"searchable": false, "filterable": false, "sortable": false, "facetable":
false, "key": true },
        { "name": "language", "type": "Edm.String", "filterable": true,
"sortable": false, "facetable": true },
        { "name": "translated_text", "type": "Edm.String", "filterable":
false, "sortable": false, "facetable": false },
        { "name": "sentiment", "type": "Collection(Edm.String)",
"searchable": false, "filterable": true, "retrievable": true, "sortable":
false, "facetable": true },
        { "name": "keyphrases", "type": "Collection(Edm.String)",
"filterable": true, "sortable": false, "facetable": true }
    ]
}
```

This index definition is a combination of data that you'd like to present to the user. It includes fields mapped directly to the CSV (name of the hotel, review date, and so forth) and fields created by the skillset (sentiment, key phrases, language, and translated text).

Select **Send** to issue the PUT request.

You should see the status `201 - Created`. If you see a different status, in the **Body** pane, look for a JSON response that contains an error message. If you are using a free search service, make sure you have room for the new objects (the free service has a maximum of three).

At this point, the index is created but not loaded. Importing documents occurs later when you run the indexer.

Create a data source

Next, connect Azure Cognitive Search to the hotel data you stored in Blob storage. To create the data source, send a Create Data Source POST request to `https://{{search-service-name}}.search.windows.net/datasources?api-version={{api-version}}`.

In Postman, go to the **Create Datasource** request, and then to the **Body** pane. You should see the following code:

JSON

```
{
  "name" : "{{datasource-name}}",
  "description" : "Demo files to demonstrate knowledge store capabilities.",
  "type" : "azureblob",
```

```
  "credentials" : { "connectionString" : "{{storage-connection-string}}" },
  "container" : { "name" : "{{storage-container-name}}" }
}
```

Select **Send** to issue the POST request.

Create a skillset

The next step creates the skillset, which specifies both the enhancements to be applied and the knowledge store where the results will be stored. This request sends a Create Skillset PUT request to `https://{{search-service-name}}.search.windows.net/skillsets/{{skillset-name}}?api-version={{api-version}}`.

There are two large top-level objects: `skills` and `knowledgeStore`:

- "skills" is the skillset. Each object inside the `skills` object is an enrichment service. Each enrichment service has `inputs` and `outputs`. The `LanguageDetectionSkill` has an output `targetName` of `Language`. The value of this node is used by most of the other skills as an input. The source is `document/Language`. The capability of using the output of one node as the input to another is even more evident in `ShaperSkill`, which specifies how the data flows into the tables of the knowledge store.

- "knowledgeStore" includes the connection string to the storage account and a series of projections. Each projection item in the projections array results in a table in Azure Storage. Columns in the table will include generated columns used for cross-linking and content fields. They will also include content fields created during enrichment.

  Projections accept data shapes as input. You can provide the shapes through a Shaper skill, mapping the skill output to a projection, or you can define inline shapes in each projection item.

To generate the skillset, select the **Send** button in Postman to PUT the request:

JSON

```
{
    "name": "{{skillset-name}}",
    "description": "Skillset to detect language, translate text, extract key phrases, and score sentiment",
    "skills": [
        {
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
```

```
            "context": "/document/reviews_text", "textSplitMode": "pages",
"maximumPageLength": 5000,
            "inputs": [
                { "name": "text", "source": "/document/reviews_text" }
            ],
            "outputs": [
                { "name": "textItems", "targetName": "pages" }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.V3.SentimentSkill",
            "context": "/document/reviews_text/pages/*",
            "inputs": [
                { "name": "text", "source": "/document/reviews_text/pages/*"
},
                { "name": "languageCode", "source": "/document/language" }
            ],
            "outputs": [
                { "name": "sentiment", "targetName": "sentiment" }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
            "context": "/document",
            "inputs": [
                { "name": "text", "source": "/document/reviews_text" }
            ],
            "outputs": [
                { "name": "languageCode", "targetName": "language" }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",
            "context": "/document/reviews_text/pages/*",
            "defaultFromLanguageCode": null,
            "defaultToLanguageCode": "en",
            "inputs": [
                { "name": "text", "source": "/document/reviews_text/pages/*"
}
            ],
            "outputs": [
                { "name": "translatedText", "targetName": "translated_text"
}
            ]
        },
        {
            "@odata.type":
"#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
            "context": "/document/reviews_text/pages/*",
            "inputs": [
                { "name": "text",  "source":
"/document/reviews_text/pages/*" },
                { "name": "languageCode",  "source": "/document/language" }
            ],
            "outputs": [
```

```
                { "name": "keyPhrases" , "targetName": "keyphrases" }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
            "context": "/document",
            "inputs": [
                { "name": "name",  "source": "/document/name" },
                { "name": "reviews_date",  "source":
"/document/reviews_date" },
                { "name": "reviews_rating",  "source":
"/document/reviews_rating" },
                { "name": "reviews_text",  "source":
"/document/reviews_text" },
                { "name": "reviews_title",  "source":
"/document/reviews_title" },
                { "name": "reviews_username",  "source":
"/document/reviews_username" },
                { "name": "AzureSearch_DocumentKey",  "source":
"/document/AzureSearch_DocumentKey" },
                {
                "name": "pages",
                "sourceContext": "/document/reviews_text/pages/*",
                "inputs": [
                    {
                    "name": "languageCode",
                    "source": "/document/language"
                    },
                    {
                    "name": "translatedText",
                    "source":
"/document/reviews_text/pages/*/translated_text"
                    },
                    {
                    "name": "sentiment",
                    "source": "/document/reviews_text/pages/*/sentiment"
                    },
                    {
                    "name": "keyPhrases",
                    "source": "/document/reviews_text/pages/*/keyphrases/*"
                    },
                    {
                    "name": "Page",
                    "source": "/document/reviews_text/pages/*"
                    }
                ]
                }
            ],
            "outputs": [
                { "name": "output" , "targetName": "tableprojection" }
            ]
        }
    ],
    "knowledgeStore": {
        "storageConnectionString": "{{storage-connection-string}}",
```

```
        "projections": [
            {
                "tables": [
                    { "tableName": "hotelReviews1Document",
"generatedKeyName": "Documentid", "source": "/document/tableprojection" },
                    { "tableName": "hotelReviews2Pages", "generatedKeyName":
"Pagesid", "source": "/document/tableprojection/pages/*" },
                    { "tableName": "hotelReviews3KeyPhrases",
"generatedKeyName": "KeyPhrasesid", "source":
"/document/tableprojection/pages/*/keyPhrases/*" }
                ],
                "objects": []
            },
            {
                "tables": [
                    {
                        "tableName":
"hotelReviews4InlineProjectionDocument", "generatedKeyName": "Documentid",
"sourceContext": "/document",
                        "inputs": [
                            { "name": "name", "source": "/document/name"},
                            { "name": "reviews_date", "source":
"/document/reviews_date"},
                            { "name": "reviews_rating", "source":
"/document/reviews_rating"},
                            { "name": "reviews_username", "source":
"/document/reviews_username"},
                            { "name": "reviews_title", "source":
"/document/reviews_title"},
                            { "name": "reviews_text", "source":
"/document/reviews_text"},
                            { "name": "AzureSearch_DocumentKey", "source":
"/document/AzureSearch_DocumentKey" }
                        ]
                    },
                    {
                        "tableName": "hotelReviews5InlineProjectionPages",
"generatedKeyName": "Pagesid", "sourceContext":
"/document/reviews_text/pages/*",
                        "inputs": [
                            { "name": "Sentiment", "source":
"/document/reviews_text/pages/*/sentiment"},
                            { "name": "LanguageCode", "source":
"/document/language"},
                            { "name": "Keyphrases", "source":
"/document/reviews_text/pages/*/keyphrases"},
                            { "name": "TranslatedText", "source":
"/document/reviews_text/pages/*/translated_text"},
                            { "name": "Page", "source":
"/document/reviews_text/pages/*" }
                        ]
                    },
                    {
                        "tableName":
"hotelReviews6InlineProjectionKeyPhrases", "generatedKeyName": "kpidv2",
```

```
"sourceContext": "/document/reviews_text/pages/*/keyphrases/*",
                        "inputs": [
                            { "name": "Keyphrases", "source":
"/document/reviews_text/pages/*/keyphrases/*" }
                        ]
                    }
                ],
                "objects": []
            }
        ]
    }
}
```

Create an indexer

The final step is the Create Indexer request. The indexer reads the data and activates the skillset. The definition of the indexer refers to several other resources that you already created: the datasource, the index, and the skillset.

- The `parameters/configuration` object controls how the indexer ingests the data. In this case, the input data is in a single CSV file that has a header line and comma-separated values.
- Field mappings create "AzureSearch_DocumentKey" is a unique identifier for each document that's generated by the blob indexer (based on metadata storage path).
- Output field mappings specify how enriched fields are mapped to fields in a search index. Output field mappings are not used in knowledge stores (knowledge stores use shapes and projections to express the physical data structures).

Select **Send** in Postman to create and run the indexer. Data import, skillset execution, and knowledge store creation occur in this step.

JSON

```
{
    "name": "{{indexer-name}}",
    "dataSourceName": "{{datasource-name}}",
    "skillsetName": "{{skillset-name}}",
    "targetIndexName": "{{index-name}}",
    "parameters": {
        "configuration": {
            "dataToExtract": "contentAndMetadata",
            "parsingMode": "delimitedText",
            "firstLineContainsHeaders": true,
            "delimitedTextDelimiter": ","
        }
    },
```

```
    "fieldMappings": [
        {
            "sourceFieldName": "AzureSearch_DocumentKey",
            "targetFieldName": "AzureSearch_DocumentKey",
            "mappingFunction": { "name": "base64Encode" }
        }
    ],
    "outputFieldMappings": [
        { "sourceFieldName": "/document/reviews_text/pages/*/Keyphrases/*",
"targetFieldName": "Keyphrases" },
        { "sourceFieldName": "/document/Language", "targetFieldName":
"Language" },
        { "sourceFieldName": "/document/reviews_text/pages/*/Sentiment",
"targetFieldName": "Sentiment" }
    ]
}
```

Check status

After you send each request, the search service should respond with a 201 success message. If you get errors, re-check your variables and make sure that the search service has room for the new index, indexer, data source, and skillset (the free tier is limited to three of each).

In the Azure portal, go to the Azure Cognitive Search service's **Overview** page. Select the **Indexers** tab, and then select **hotels-reviews-ixr**. Within a minute or two, status should progress from "In progress" to "Success" with zero errors and warnings.

Check tables in Azure portal

In the Azure portal, switch to your Azure Storage account and use **Storage Browser** to view the new tables. You should see six tables, one for each projection defined in the skillset.

Each table is generated with the IDs necessary for cross-linking the tables in queries. When you open a table, scroll past these fields to view the content fields added by the pipeline.

In this walkthrough, the knowledge store is composed of a various tables showing different ways of shaping and structuring a table. Tables one through three use output from a Shaper skill to determine the columns and rows. Tables four through six are created from inline shaping instructions, embedded within the projection itself. You can use either approach to achieve the same outcome.

Table	Description
hotelReviews1Document	Contains fields carried forward from the CSV, such as reviews_date and reviews_text.
hotelReviews2Pages	Contains enriched fields created by the skillset, such as sentiment score and translated text.
hotelReviews3KeyPhrases	Contains a long list of just the key phrases.
hotelReviews4InlineProjectionDocument	Alternative to the first table, using inline shaping instead of the Shaper skill to shape data for the projection.
hotelReviews5InlineProjectionPages	Alternative to the second table, using inline shaping.
hotelreviews6InlineProjectionKeyPhrases	Alternative to the third table, using inline shaping.

Clean up

When you're working in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources in the portal, using the **All resources** or **Resource groups** link in the left-navigation pane.

If you are using a free service, remember that you are limited to three indexes, indexers, and data sources. You can delete individual items in the portal to stay under the limit.

> **Tip**
>
> If you want to repeat this exercise or try a different AI enrichment walkthrough, delete the **hotel-reviews-idxr** indexer and the related objects to recreate them. Deleting the indexer resets the free daily transaction counter to zero.

Next steps

Now that you've enriched your data by using Cognitive Services and projected the results to a knowledge store, you can use Storage Explorer or other apps to explore your enriched data set.

Get started with Storage Explorer

Shaping data for projection into a knowledge store

Article • 01/31/2023 • 5 minutes to read

In Azure Cognitive Search, "shaping data" describes a step in the knowledge store workflow that creates a data representation of the content that you want to project into tables, objects, and files in Azure Storage.

As skills execute, the outputs are written to an enrichment tree in a hierarchy of nodes, and while you might want to view and consume the enrichment tree in its entirety, it's more likely that you'll want a finer grain, creating subsets of nodes for different scenarios, such as placing the nodes related to translated text or extracted entities in specific tables.

By itself, the enrichment tree doesn't include logic that would inform how its content is represented in a knowledge store. Data shapes fill this gap by providing the schema of what goes into each table, object, and file projection. You can think of a data shape as a custom definition or view of the enriched data. You can create as many shapes as you need, and then assign them to projections in a knowledge store definition.

Approaches for creating shapes

There are two ways to shape enriched content to that it can be projected into a knowledge store:

- Use the Shaper skill to create nodes in an enrichment tree that are used expressly for projection. Most skills create new content. In contrast, a Shaper skill work with existing nodes, usually to consolidate multiple nodes into a single complex object. This is useful for tables, where you want the output of multiple nodes to be physically expressed as columns in the table.
- Use an inline shape within the projection definition itself.

Using the Shaper skill externalizes the shape so that it can be used by multiple projections or even other skills. It also ensures that all the mutations of the enrichment tree are contained within the skill, and that the output is an object that can be reused. In contrast, inline shaping allows you to create the shape you need, but is an anonymous object and is only available to the projection for which it's defined.

The approaches can be used together or separately. This article shows both: a Shaper skill for the table projections, and inline shaping with the key phrases table projection.

Use a Shaper skill

Shaper skills are usually placed at the end of a skillset, creating a view of the data that you want to pass to a projection. This example creates a shape called "tableprojection" containing the following nodes: "reviews_text", "reviews_title", "AzureSearch_DocumentKey", and sentiment scores and key phrases from paged reviews.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "#5",
    "description": null,
    "context": "/document",
    "inputs": [
        {
            "name": "reviews_text",
            "source": "/document/reviews_text",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "reviews_title",
            "source": "/document/reviews_title",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "AzureSearch_DocumentKey",
            "source": "/document/AzureSearch_DocumentKey",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "pages",
            "source": null,
            "sourceContext": "/document/reviews_text/pages/*",
            "inputs": [
                {
                    "name": "Sentiment",
                    "source": "/document/reviews_text/pages/*/Sentiment",
                    "sourceContext": null,
                    "inputs": []
                },
                {
                    "name": "LanguageCode",
                    "source": "/document/Language",
                    "sourceContext": null,
                    "inputs": []
                },
                {
```

```
                    "name": "Page",
                    "source": "/document/reviews_text/pages/*",
                    "sourceContext": null,
                    "inputs": []
                },
                {
                    "name": "keyphrase",
                    "sourceContext":
"/document/reviews_text/pages/*/Keyphrases/*",
                    "inputs": [
                        {
                            "source":
"/document/reviews_text/pages/*/Keyphrases/*",
                            "name": "Keyphrases"
                        }
                    ]
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "tableprojection"
        }
    ]
}
```

SourceContext property

Within a Shaper skill, an input can have a `sourceContext` element. This same property can also be used in inline shapes in projections.

`sourceContext` is used to construct multi-level, nested objects in an enrichment pipeline. If the input is at a *different* context than the skill context, use the *sourceContext*. The *sourceContext* requires you to define a nested input with the specific element being addressed as the source.

In the example above, sentiment analysis and key phrases extraction was performed on text that was split into pages for more efficient analysis. Assuming you want the scores and phrases projected into a table, you'll now need to set the context to nested input that provides the score and phrase.

Projecting a shape into multiple tables

With the `tableprojection` node defined in the `outputs` section above, you can slice parts of the `tableprojection` node into individual, related tables:

JSON

```
"projections": [
    {
        "tables": [
            {
                "tableName": "hotelReviewsDocument",
                "generatedKeyName": "Documentid",
                "source": "/document/tableprojection"
            },
            {
                "tableName": "hotelReviewsPages",
                "generatedKeyName": "Pagesid",
                "source": "/document/tableprojection/pages/*"
            },
            {
                "tableName": "hotelReviewsKeyPhrases",
                "generatedKeyName": "KeyPhrasesid",
                "source": "/document/tableprojection/pages/*/keyphrase/*"
            }
        ]
    }
]
```

Inline shape for table projections

Inline shaping is the ability to form new shapes within the projection definition itself. Inline shaping has these characteristics:

- The shape can be used only by the projection that contains it.
- The shape can be identical to what a Shaper skill would produce.

An inline shape is created using `sourceContext` and `inputs`.

Property	Description
sourceContext	Sets the root of the projection.
inputs	Each input is a column in the table. Name is the column name. Source is the enrichment node that provides the value.

To project the same data as the previous example, the inline projection option would look like this:

JSON

```
"projections": [
    {
```

```
        "tables": [
            {
                "tableName": "hotelReviewsInlineDocument",
                "generatedKeyName": "Documentid",
                "sourceContext": "/document",
                "inputs": [
                    {
                        "name": "reviews_text",
                        "source": "/document/reviews_text"
                    },
                    {
                        "name": "reviews_title",
                        "source": "/document/reviews_title"
                    },
                    {
                        "name": "AzureSearch_DocumentKey",
                        "source": "/document/AzureSearch_DocumentKey"
                    }
                ]
            },
            {
                "tableName": "hotelReviewsInlinePages",
                "generatedKeyName": "Pagesid",
                "sourceContext": "/document/reviews_text/pages/*",
                "inputs": [
                        {
                    "name": "Sentiment",
                    "source": "/document/reviews_text/pages/*/Sentiment"
                    },
                    {
                        "name": "LanguageCode",
                        "source": "/document/Language"
                    },
                    {
                        "name": "Page",
                        "source": "/document/reviews_text/pages/*"
                    }
                ]
            },
            {
                "tableName": "hotelReviewsInlineKeyPhrases",
                "generatedKeyName": "KeyPhraseId",
                "sourceContext":
"/document/reviews_text/pages/*/Keyphrases/*",
                "inputs": [
                    {
                        "name": "Keyphrases",
                        "source":
"/document/reviews_text/pages/*/Keyphrases/*"
                    }
                ]
            }
        ]
    }
]
```

One observation from both the approaches is how values of "Keyphrases" are projected using the "sourceContext". The "Keyphrases" node, which contains a collection of strings, is itself a child of the page text. However, because projections require a JSON object and the page is a primitive (string), the "sourceContext" is used to wrap the key phrase into an object with a named property. This technique enables even primitives to be projected independently.

Inline shape for object projections

You can generate a new shape using the Shaper skill or use inline shaping of the object projection. While the tables example demonstrated the approach of creating a shape and slicing, this example demonstrates the use of inline shaping.

Inline shaping is the ability to create a new shape in the definition of the inputs to a projection. Inline shaping creates an anonymous object that is identical to what a Shaper skill would produce (in this case, `projectionShape`). Inline shaping is useful if you're defining a shape that you don't plan to reuse.

The projections property is an array. This example adds a new projection instance to the array, where the knowledgeStore definition contains inline projections. When using inline projections, you can omit the Shaper skill.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
            {
            "tables": [ ],
            "objects": [
                {
                    "storageContainer": "sampleobject",
                    "source": null,
                    "generatedKeyName": "myobject",
                    "sourceContext": "/document",
                    "inputs": [
                        {
                            "name": "metadata_storage_name",
                            "source": "/document/metadata_storage_name"
                        },
                        {
                            "name": "metadata_storage_path",
                            "source": "/document/metadata_storage_path"
                        },
                        {
```

```
                            "name": "content",
                            "source": "/document/content"
                        },
                        {
                            "name": "keyPhrases",
                            "source":
"/document/merged_content/keyphrases/*"
                        },
                        {
                            "name": "entities",
                            "source":
"/document/merged_content/entities/*/name"
                        },
                        {
                            "name": "ocrText",
                            "source": "/document/normalized_images/*/text"
                        },
                        {
                            "name": "ocrLayoutText",
                            "source":
"/document/normalized_images/*/layoutText"
                        }
                    ]

                }
            ],
            "files": []
        }
    ]
}
```

Next steps

This article describes the concepts and principles of projection shapes. As a next step, see how these are applied in patterns for table, object, and file projections.

Define projections in a knowledge store

Additional resources

Documentation

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Text Merge cognitive skill - Azure Cognitive Search

Merge text from a collection of fields into one consolidated field. Use this cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Create a knowledge store using REST - Azure Cognitive Search

Use the REST API and Postman to create an Azure Cognitive Search knowledge store for persisting AI enrichments from skillset.

Show 5 more

Define projections in a knowledge store

Article • 01/31/2023 • 9 minutes to read

Projections are the component of a knowledge store definition that determines where AI enriched content is stored in Azure Storage. Projections determine the type, quantity, and composition of the data structures containing your content.

In this article, learn the syntax for each type of projection:

- Table projections
- Object projections
- File projections

Recall that projections are defined under the "knowledgeStore" property of a skillset.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
      {
        "tables": [ ],
        "objects": [ ],
        "files": [ ]
      }
    ]
```

If you need more background before getting started, review this check list for tips and workflow.

> **Tip**
>
> When developing projections, **enable enrichment caching (preview)** so that you can reuse existing enrichments while editing projection definitions. Enrichment caching is a preview feature, so be sure to use the preview REST API (api-version=2020-06-30-preview or later) on the indexer request. Without caching, simple edits to a projection will result in a full reprocess of enriched content. By caching the enrichments, you can iterate over projections without incurring any skillset processing charges.

Requirements

All projections have source and destination properties. The source is always internal content from an enrichment tree created during skillset execution. The destination is the name and type of an external object that's created and populated in Azure Storage.

Except for file projections, which only accept binary images, the source must be:

- Valid JSON
- A path to a node in the enrichment tree (for example, `"source": "/document/objectprojection"`)

While a node might resolve to a single field, a more common representation is a reference to a complex shape. Complex shapes are created through a shaping methodology, either a Shaper skill or an inline shaping definition, but usually a Shaper skill. The fields or elements of the shape determine the fields in containers and tables.

Shaper skills are favored because it outputs JSON, where as most skills don't output valid JSON on their own. In many cases, the same data shape created by a Shaper skill can be used equally by both table and object projections.

Given source input requirements, knowing how to shape data becomes a practical requirement for projection definition, especially if you're working with tables.

Define a table projection

Table projections are recommended for scenarios that call for data exploration, such as analysis with Power BI or workloads that consume data frames. The tables section of a projections array is a list of tables that you want to project.

To define a table projection, use the `tables` array in the projections property. A table projection has three required properties:

Property	Description
tableName	Determines the name of a new table created in Azure Table Storage.
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column will be created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree. The node should be a reference to a complex shape that determines which columns are created in the table.

In table projections, "source" is usually the output of a Shaper skill that defines the shape of the table. Tables have rows and columns, and shaping is the mechanism by

which rows and columns are specified. You can use a Shaper skill or inline shapes. The Shaper skill produces valid JSON, but the source could be the output from any skill, if valid JSON.

> **Note**
>
> Table projections are subject to the **storage limits** imposed by Azure Storage. The entity size cannot exceed 1 MB and a single property can be no bigger than 64 KB. These constraints make tables a good solution for storing a large number of small entities.

Single table example

The schema of a table is specified partly by the projection (table name and key), and also by the source that provides the shape of table (columns). This example shows just one table so that you can focus on the details of the definition.

JSON

```
"projections" : [
  {
    "tables": [
      { "tableName": "Hotels", "generatedKeyName": "HotelId", "source":
"/document/tableprojection" }
    ]
  }
]
```

Columns are derived from the "source". The following data shape containing HotelId, HotelName, Category, and Description will result in creation of those columns in the table.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "#3",
    "description": null,
    "context": "/document",
    "inputs": [
    {
        "name": "HotelId",
        "source": "/document/HotelId"
    },
    {
        "name": "HotelName",
```

```
        "source": "/document/HotelName"
    },
    {
        "name": "Category",
        "source": "/document/Category"
    },
    {
        "name": "Description",
        "source": "/document/Description"
    },
    ],
    "outputs": [
    {
        "name": "output",
        "targetName": "tableprojection"
    }
    ]
}
```

Multiple table (slicing) example

A common pattern for table projections is to have multiple related tables, where system-generated partitionKey and rowKey columns are created to support cross-table relationships for all tables under the same projection group.

Creating multiple tables can be useful if you want control over how related data is aggregated. If enriched content has unrelated or independent components, for example the keywords extracted from a document might be unrelated from the entities recognized in the same document, you can split those fields out into adjacent tables.

When you're projecting to multiple tables, the complete shape is projected into each table, unless a child node is the source of another table within the same group. Adding a projection with a source path that is a child of an existing projection results in the child node being sliced out of the parent node and projected into the new yet related table. This technique allows you to define a single node in a Shaper skill that can be the source for all of your projections.

The pattern for multiple tables consists of:

- One table as the parent or main table
- Additional tables to contain slices of the enriched content

For example, assume a Shaper skill outputs an "EnrichedShape" that contains hotel information, plus enriched content like key phrases, locations, and organizations. The main table would include fields that describe the hotel (ID, name, description, address,

category). Key phrases would get the key phrase column. Entities would get the entity columns.

JSON

```
"projections" : [
  {
    "tables": [
    { "tableName": "MainTable", "generatedKeyName": "HotelId", "source":
"/document/EnrichedShape" },
    { "tableName": "KeyPhrases", "generatedKeyName": "KeyPhraseId",
"source": "/document/EnrichedShape/*/KeyPhrases/*" },
    { "tableName": "Entities", "generatedKeyName": "EntityId", "source":
"/document/EnrichedShape/*/Entities/*" }
    ]
  }
]
```

Naming relationships

The `generatedKeyName` and `referenceKeyName` properties are used to relate data across tables or even across projection types. Each row in the child table has a property pointing back to the parent. The name of the column or property in the child is the `referenceKeyName` from the parent. When the `referenceKeyName` isn't provided, the service defaults it to the `generatedKeyName` from the parent.

Power BI relies on these generated keys to discover relationships within the tables. If you need the column in the child table named differently, set the `referenceKeyName` property on the parent table. One example would be to set the `generatedKeyName` as ID on the tblDocument table and the `referenceKeyName` as DocumentID. This would result in the column in the tblEntities and tblKeyPhrases tables containing the document ID being named DocumentID.

Define an object projection

Object projections are JSON representations of the enrichment tree that can be sourced from any node. In comparison with table projections, object projections are simpler to define and are used when projecting whole documents. Object projections are limited to a single projection in a container and can't be sliced.

To define an object projection, use the `objects` array in the projections property. An object projection has three required properties:

Property	Description
storageContainer	Determines the name of a new container created in Azure Storage.
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column will be created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree that is the root of the projection. The node is usually a reference to a complex data shape that determines blob structure.

The following example projects individual hotel documents, one hotel document per blob, into a container called `hotels`.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "an Azure storage connection string",
  "projections" : [
    {
      "tables": [ ]
    },
    {
      "objects": [
        {
        "storageContainer": "hotels",
        "source": "/document/objectprojection",
        }
      ]
    },
    {
        "files": [ ]
    }
  ]
}
```

The source is the output of a Shaper skill, named `"objectprojection"`. Each blob will have a JSON representation of each field input.

JSON

```
    {
      "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
      "name": "#3",
      "description": null,
      "context": "/document",
      "inputs": [
        {
```

```
            "name": "HotelId",
            "source": "/document/HotelId"
          },
          {
            "name": "HotelName",
            "source": "/document/HotelName"
          },
          {
            "name": "Category",
            "source": "/document/Category"
          },
          {
            "name": "keyPhrases",
            "source": "/document/HotelId/keyphrases/*"
          },
        ],
        "outputs": [
          {
            "name": "output",
            "targetName": "objectprojection"
          }
        ]
      }
```

Define a file projection

File projections are always binary, normalized images, where normalization refers to potential resizing and rotation for use in skillset execution. File projections, similar to object projections, are created as blobs in Azure Storage, and contain binary data (as opposed to JSON).

To define a file projection, use the `files` array in the projections property. A files projection has three required properties:

Property	Description
storageContainer	Determines the name of a new container created in Azure Storage.
generatedKeyName	Column name for the key that uniquely identifies each row. The value is system-generated. If you omit this property, a column will be created automatically that uses the table name and "key" as the naming convention.
source	A path to a node in an enrichment tree that is the root of the projection. For images files, the source is always `/document/normalized_images/*`. File projections only act on the `normalized_images` collection. Neither indexers nor a skillset will pass through the original non-normalized image.

The destination is always a blob container, with a folder prefix of the base64 encoded value of the document ID. If there are multiple images, they'll be placed together in the same folder. File projections can't share the same container as object projections and need to be projected into a different container.

The following example projects all normalized images extracted from the document node of an enriched document, into a container called `myImages`.

JSON

```
"projections": [
    {
        "tables": [ ],
        "objects": [ ],
        "files": [
            {
                "storageContainer": "myImages",
                "source": "/document/normalized_images/*"
            }
        ]
    }
]
```

Test projections

You can process projections by following these steps:

1. Set the knowledge store's `storageConnectionString` property to a valid V2 general purpose storage account connection string.
2. Update the skillset by issuing a PUT request with your projection definition in the body of the skillset.
3. Run the indexer to put the skillset into execution.
4. Monitor indexer execution to check progress and catch any errors.
5. Use Azure portal to verify object creation in Azure Storage.
6. If you're projecting tables, import them into Power BI for table manipulation and visualization. In most cases, Power BI will auto-discover the relationships among tables.

Common issues

Omitting any of the following steps can result in unexpected outcomes. Check for the following conditions if your output doesn't look right.

- String enrichments aren't shaped into valid JSON. When strings are enriched, for example `merged_content` enriched with key phrases, the enriched property is represented as a child of `merged_content` within the enrichment tree. The default representation isn't well-formed JSON. At projection time, make sure to transform the enrichment into a valid JSON object with a name and a value. Using a Shaper skill or defining inline shapes will help resolve this issue.

- Omission of `/*` at the end of a source path. If the source of a projection is `/document/projectionShape/keyPhrases`, the key phrases array is projected as a single object/row. Instead, set the source path to `/document/projectionShape/keyPhrases/*` to yield a single row or object for each of the key phrases.

- Path syntax errors. Path selectors are case-sensitive and can lead to missing input warnings if you don't use the exact case for the selector.

Next steps

The next step walks you through shaping and projection of output from a rich skillset. If your skillset is complex, the following article provides examples of both shapes and projections.

Detailed example of shapes and projections

Detailed example of shapes and projections in a knowledge store

Article • 03/24/2023 • 11 minutes to read

This article provides a detailed example that supplements high-level concepts and syntax-based articles by walking you through the shaping and projection steps required for fully expressing the output of a rich skillset in a knowledge store.

If your application requirements call for multiple skills and projections, this example can give you a better idea of how shapes and projections intersect.

Download sample definitions

This example uses Postman app and the Search REST APIs.

Clone or download azure-search-postman-samples on GitHub and import the **Projections collection** to step through this example yourself.

Set up sample data

Sample documents aren't included with the Projections collection, but the AI enrichment demo data files from the azure-search-sample-data repo contain text and images, and will work with the projections described in this example.

Create a blob container in Azure Storage and upload all 14 items.

While in Azure Storage, copy a connection string so that you can specify it in the Postman collection.

Example skillset

To understand the dependency between shapes and projections, review the following skillset that creates enriched content. This skillset processes both raw images and text, producing outputs that will be referenced in shapes and projections.

Pay close attention to skill outputs (targetNames). Outputs written to the enriched document tree are referenced in projections and in shapes (via Shaper skills).

JSON

```
{
    "name": "projections-demo-ss",
    "description": "Skillset that enriches blob data found in
"merged_content". The enrichment granularity is a document.",
    "skills": [
        {
            "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
            "name": "#1",
            "description": null,
            "context": "/document/merged_content",
            "categories": [
                "Person",
                "Quantity",
                "Organization",
                "URL",
                "Email",
                "Location",
                "DateTime"
            ],
            "defaultLanguageCode": "en",
            "minimumPrecision": null,
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/merged_content"
                },
                {
                    "name": "languageCode",
                    "source": "/document/language"
                }
            ],
            "outputs": [
                {
                    "name": "persons",
                    "targetName": "people"
                },
                {
                    "name": "organizations",
                    "targetName": "organizations"
                },
                {
                    "name": "locations",
                    "targetName": "locations"
                }
            ]
        },
        {
            "@odata.type":
"#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
            "name": "#2",
            "description": null,
            "context": "/document/merged_content",
            "defaultLanguageCode": "en",
```

```
            "maxKeyPhraseCount": null,
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/merged_content"
                },
                {
                    "name": "languageCode",
                    "source": "/document/language"
                }
            ],
            "outputs": [
                {
                    "name": "keyPhrases",
                    "targetName": "keyphrases"
                }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
            "name": "#3",
            "description": null,
            "context": "/document",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/merged_content"
                }
            ],
            "outputs": [
                {
                    "name": "languageCode",
                    "targetName": "language"
                }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
            "name": "#4",
            "description": null,
            "context": "/document",
            "insertPreTag": " ",
            "insertPostTag": " ",
            "inputs": [
                {
                    "name": "text",
                    "source": "/document/content"
                },
                {
                    "name": "itemsToInsert",
                    "source": "/document/normalized_images/*/text"
                },
                {
                    "name": "offsets",
                    "source": "/document/normalized_images/*/contentOffset"
```

```
                }
            ],
            "outputs": [
                {
                    "name": "mergedText",
                    "targetName": "merged_content"
                }
            ]
        },
        {
            "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
            "name": "#5",
            "description": null,
            "context": "/document/normalized_images/*",
            "textExtractionAlgorithm": "printed",
            "lineEnding": "Space",
            "defaultLanguageCode": "en",
            "detectOrientation": true,
            "inputs": [
                {
                    "name": "image",
                    "source": "/document/normalized_images/*"
                }
            ],
            "outputs": [
                {
                    "name": "text",
                    "targetName": "text"
                },
                {
                    "name": "layoutText",
                    "targetName": "layoutText"
                }
            ]
        }
    ],
    "cognitiveServices": {
        "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",
        "description": "A Cognitive Services resource in the same region as
Search.",
        "key": "<COGNITIVE SERVICES All-in-ONE KEY>"
    },
    "knowledgeStore": null
}
```

Example Shaper skill

A Shaper skill is a utility for working with existing enriched content instead of creating new enriched content. Adding a Shaper to a skillset lets you create a custom shape that you can project into table or blob storage. Without a custom shape, projections are limited to referencing a single node (one projection per output), which isn't suitable for

tables. Creating a custom shape aggregates various elements into a new logical whole that can be projected as a single table, or sliced and distributed across a collection of tables.

In this example, the custom shape combines blob metadata and identified entities and key phrases. The custom shape is called `projectionShape` and is parented under `/document`.

One purpose of shaping is to ensure that all enrichment nodes are expressed in well-formed JSON, which is required for projecting into knowledge store. This is especially true when an enrichment tree contains nodes that aren't well-formed JSON (for example, when an enrichment is parented to a primitive like a string).

Notice the last two nodes, `KeyPhrases` and `Entities`. These are wrapped into a valid JSON object with the `sourceContext`. This is required as `keyphrases` and `entities` are enrichments on primitives and need to be converted to valid JSON before they can be projected.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "ShaperForTables",
    "description": null,
    "context": "/document",
    "inputs": [
        {
            "name": "metadata_storage_content_type",
            "source": "/document/metadata_storage_content_type",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "metadata_storage_name",
            "source": "/document/metadata_storage_name",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "metadata_storage_path",
            "source": "/document/metadata_storage_path",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "metadata_content_type",
            "source": "/document/metadata_content_type",
            "sourceContext": null,
            "inputs": []
        },
```

```
        {
            "name": "keyPhrases",
            "source": null,
            "sourceContext": "/document/merged_content/keyphrases/*",
            "inputs": [
                {
                    "name": "KeyPhrases",
                    "source": "/document/merged_content/keyphrases/*"
                }

            ]
        },
        {
            "name": "Entities",
            "source": null,
            "sourceContext": "/document/merged_content/entities/*",
            "inputs": [
                {
                    "name": "Entities",
                    "source": "/document/merged_content/entities/*/name"
                }

            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "projectionShape"
        }
    ]
}
```

Add Shapers to a skillset

The example skillset introduced at the start of this article didn't include the Shaper skill, but Shaper skills belong in a skillset and are often placed towards the end.

Within a skillset, a Shaper skill might look like this:

JSON

```
    "name": "projections-demo-ss",
    "skills": [
        {
            <Shaper skill goes here>
            }
        ],
    "cognitiveServices":  "A key goes here",
    "knowledgeStore": []
}
```

Projecting to tables

Drawing on the examples above, there's a known quantity of enrichments and data shapes that can be referenced in table projections. In the tables projection below, three tables are defined by setting the `tableName`, `source` and `generatedKeyName` properties.

All three of these tables will be related through generated keys and by the shared parent `/document/projectionShape`.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "tblDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/projectionShape"
                },
                {
                    "tableName": "tblKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/projectionShape/keyPhrases/*"
                },
                {
                    "tableName": "tblEntities",
                    "generatedKeyName": "Entityid",
                    "source": "/document/projectionShape/Entities/*"
                }
            ],
            "objects": [],
            "files": []
        }
    ]
}
```

Test your work

You can check projection definitions by following these steps:

1. Set the knowledge store's `storageConnectionString` property to a valid V2 general purpose storage account connection string.

2. Update the skillset by issuing the PUT request.

3. After updating the skillset, run the indexer.

You now have a working projection with three tables. Importing these tables into Power BI should result in Power BI discovering the relationships.

Before moving on to the next example, let's revisit aspects of the table projection to understand the mechanics of slicing and relating data.

Slicing a table into multiple child tables

Slicing is a technique that subdivides a whole consolidated shape into constituent parts. The outcome consists of separate but related tables that you can work with individually.

In the example, `projectionShape` is the consolidated shape (or enrichment node). In the projection definition, `projectionShape` is sliced into additional tables, which enables you to pull out parts of the shape, `keyPhrases` and `Entities`. In Power BI, this is useful as multiple entities and keyPhrases are associated with each document, and you'll get more insights if you can see entities and keyPhrases as categorized data.

Slicing implicitly generates a relationship between the parent and child tables, using the `generatedKeyName` in the parent table to create a column with the same name in the child table.

Naming relationships

The `generatedKeyName` and `referenceKeyName` properties are used to relate data across tables or even across projection types. Each row in the child table has a property pointing back to the parent. The name of the column or property in the child is the `referenceKeyName` from the parent. When the `referenceKeyName` isn't provided, the service defaults it to the `generatedKeyName` from the parent.

Power BI relies on these generated keys to discover relationships within the tables. If you need the column in the child table named differently, set the `referenceKeyName` property on the parent table. One example would be to set the `generatedKeyName` as ID on the tblDocument table and the `referenceKeyName` as DocumentID. This would result in the column in the tblEntities and tblKeyPhrases tables containing the document ID being named DocumentID.

Projecting blob documents

Object projections are JSON representations of the enrichment tree that can be sourced from any node. In comparison with table projections, object projections are simpler to define and are used when projecting whole documents. Object projections are limited to a single projection in a container and can't be sliced.

To define an object projection, use the `objects` array in the projections property.

The source is the path to a node of the enrichment tree that is the root of the projection. Although it isn't required, the node path is usually the output of a Shaper skill. This is because most skills don't output valid JSON objects on their own, which means that some form of shaping is necessary. In many cases, the same Shaper skill that creates a table projection can be used to generate an object projection. Alternatively, the source can also be set to a node with an inline shaping to provide the structure.

The destination is always a blob container.

The following example projects individual hotel documents, one hotel document per blob, into a container called `hotels`.

JSON

```
"knowledgeStore": {
  "storageConnectionString": "an Azure storage connection string",
  "projections" : [
    {
      "tables": [ ]
    },
    {
      "objects": [
        {
        "storageContainer": "hotels",
        "source": "/document/objectprojection",
        }
      ]
    },
    {
        "files": [ ]
    }
  ]
}
```

The source is the output of a Shaper skill, named "objectprojection". Each blob will have a JSON representation of each field input.

JSON

```
    {
      "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
```

```
      "name": "#3",
      "description": null,
      "context": "/document",
      "inputs": [
        {
          "name": "HotelId",
          "source": "/document/HotelId"
        },
        {
          "name": "HotelName",
          "source": "/document/HotelName"
        },
        {
          "name": "Category",
          "source": "/document/Category"
        },
        {
          "name": "keyPhrases",
          "source": "/document/HotelId/keyphrases/*"
        },
      ],
      "outputs": [
        {
          "name": "output",
          "targetName": "objectprojection"
        }
      ]
    }
```

Projecting an image file

File projections are always binary, normalized images, where normalization refers to potential resizing and rotation for use in skillset execution. File projections, similar to object projections, are created as blobs in Azure Storage, and contain the image.

To define a file projection, use the `files` array in the projections property.

The source is always `/document/normalized_images/*`. File projections only act on the `normalized_images` collection. Neither indexers nor a skillset will pass through the original non-normalized image.

The destination is always a blob container, with a folder prefix of the base64 encoded value of the document ID. File projections can't share the same container as object projections and need to be projected into a different container.

The following example projects all normalized images extracted from the document node of an enriched document, into a container called `myImages`.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [ ],
            "objects": [ ],
            "files": [
                {
                    "storageContainer": "myImages",
                    "source": "/document/normalized_images/*"
                }
            ]
        }
    ]
}
```

Projecting to multiple types

A more complex scenario might require you to project content across projection types. For example, projecting key phrases and entities to tables, saving OCR results of text and layout text as objects, and then projecting the images as files.

Steps for multiple projection types:

1. Create a table with a row for each document.
2. Create a table related to the document table with each key phrase identified as a row in this table.
3. Create a table related to the document table with each entity identified as a row in this table.
4. Create an object projection with the layout text for each image.
5. Create a file projection, projecting each extracted image.
6. Create a cross-reference table that contains references to the document table, object projection with the layout text, and the file projection.

Shape data for cross-projection

To get the shapes needed for these projections, start by adding a new Shaper skill that creates a shaped object called `crossProjection`.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "ShaperForCrossProjection",
    "description": null,
    "context": "/document",
    "inputs": [
        {
            "name": "metadata_storage_name",
            "source": "/document/metadata_storage_name",
            "sourceContext": null,
            "inputs": []
        },
        {
            "name": "keyPhrases",
            "source": null,
            "sourceContext": "/document/merged_content/keyphrases/*",
            "inputs": [
                {
                    "name": "KeyPhrases",
                    "source": "/document/merged_content/keyphrases/*"
                }

            ]
        },
        {
            "name": "entities",
            "source": null,
            "sourceContext": "/document/merged_content/entities/*",
            "inputs": [
                {
                    "name": "Entities",
                    "source": "/document/merged_content/entities/*/name"
                }

            ]
        },
        {
            "name": "images",
            "source": null,
            "sourceContext": "/document/normalized_images/*",
            "inputs": [
                {
                    "name": "image",
                    "source": "/document/normalized_images/*"
                },
                {
                    "name": "layoutText",
                    "source": "/document/normalized_images/*/layoutText"
                },
                {
                    "name": "ocrText",
                    "source": "/document/normalized_images/*/text"
                }
                ]
```

```
        }

    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "crossProjection"
        }
    ]
}
```

Define table, object, and file projections

From the consolidated crossProjection object, slice the object into multiple tables, capture the OCR output as blobs, and then save the image as files (also in Blob storage).

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
            {
            "tables": [
                {
                    "tableName": "crossDocument",
                    "generatedKeyName": "Id",
                    "source": "/document/crossProjection"
                },
                {
                    "tableName": "crossEntities",
                    "generatedKeyName": "EntityId",
                    "source": "/document/crossProjection/entities/*"
                },
                {
                    "tableName": "crossKeyPhrases",
                    "generatedKeyName": "KeyPhraseId",
                    "source": "/document/crossProjection/keyPhrases/*"
                },
                {
                    "tableName": "crossReference",
                    "generatedKeyName": "CrossId",
                    "source": "/document/crossProjection/images/*"
                }

            ],
            "objects": [
                {
                    "storageContainer": "crossobject",
                    "generatedKeyName": "crosslayout",
                    "source": null,
                    "sourceContext":
```

```
"/document/crossProjection/images/*/layoutText",
                    "inputs": [
                        {
                            "name": "OcrLayoutText",
                            "source":
"/document/crossProjection/images/*/layoutText"
                        }
                    ]
                }
            ],
            "files": [
                {
                    "storageContainer": "crossimages",
                    "generatedKeyName": "crossimages",
                    "source": "/document/crossProjection/images/*/image"
                }
            ]
        }
    ]
}
```

Object projections require a container name for each projection. Object projections and file projections can't share a container.

Relationships among table, object, and file projections

This example also highlights another feature of projections. By defining multiple types of projections within the same projection object, there's a relationship expressed within and across the different types (tables, objects, files). This allows you to start with a table row for a document and find all the OCR text for the images within that document in the object projection.

If you don't want the data related, define the projections in different projection groups. For example, the following snippet will result in the tables being related, but without relationships between the tables and the object (OCR text) projections.

Projection groups are useful when you want to project the same data in different shapes for different needs. For example, a projection group for the Power BI dashboard, and another projection group for capturing data used to train a machine learning model wrapped in a custom skill.

When building projections of different types, file and object projections are generated first, and the paths are added to the tables.

JSON

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=
<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "unrelatedDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/projectionShape"
                },
                {
                    "tableName": "unrelatedKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/projectionShape/keyPhrases"
                }
            ],
            "objects": [

            ],
            "files": []
        },
        {
            "tables": [],
            "objects": [
                {
                    "storageContainer": "unrelatedocrtext",
                    "source": null,
                    "sourceContext": "/document/normalized_images/*/text",
                    "inputs": [
                        {
                            "name": "ocrText",
                            "source": "/document/normalized_images/*/text"
                        }
                    ]
                },
                {
                    "storageContainer": "unrelatedocrlayout",
                    "source": null,
                    "sourceContext":
"/document/normalized_images/*/layoutText",
                    "inputs": [
                        {
                            "name": "ocrLayoutText",
                            "source":
"/document/normalized_images/*/layoutText"
                        }
                    ]
                }
            ],
            "files": []
        }
    ]
}
```

Next steps

The example in this article demonstrates common patterns on how to create projections. Now that you have a good understanding of the concepts, you're better equipped to build projections for your specific scenario.

Configure caching for incremental enrichment

Connect a knowledge store with Power BI

Article • 01/31/2023 • 3 minutes to read

In this article, learn how to connect to and query a knowledge store using Power Query in the Power BI Desktop app. You can get started faster with templates, or build a custom dashboard from scratch.

A knowledge store that's composed of tables in Azure Storage work best in Power BI. If the tables contain projections from the same skillset and projection group, you can easily "join" them to build table visualizations that include fields from related tables.

Follow the steps in this article using sample data and a knowledge store as created in this portal quickstart or through Postman and REST APIs.

Connect to Azure Storage

1. Start Power BI Desktop and select **Get data**.
2. In the **Get Data** window, select **Azure**, and then select **Azure Table Storage**.
3. Select **Connect**.
4. For **Account Name or URL**, enter in your Azure Storage account name (the full URL will be created for you).
5. If prompted, enter the storage account key.

Set up tables

1. Select the checkbox next to all of the tables that were created from the same skillset, and then select **Load**.

2. On the top ribbon, select **Transform Data** to open the **Power Query Editor**.

3. Open *hotelReviewsDocument* and remove its *PartitionKey*, *RowKey*, and *Timestamp* columns. Those columns are used for table relationships in Azure Table Storage. Power BI doesn't need them. You should be left with one column named "Content" showing *Record* in each one.

4. Click the icon with opposing arrows at the upper right side of the table to expand *Content*. When the list of columns appears, select all columns. Clear columns starting with 'metadata'. Click **OK** to include the selected columns.

5. Change the data type for the following columns by clicking the ABC-123 icon at the top left of the column.

- For *content.latitude* and *Content.longitude*, select **Decimal Number**.
- For *Content.reviews_date* and *Content.reviews_dateAdded*, select **Date/Time**.

6. Open *hotelReviewsSsPages* and repeat column deletion steps, expanding *Content* to select columns from the records. There are no data type modifications for this table.

7. Open *hotelReviewsSsKeyPhrases* and repeat column deletion steps, expanding *Content* to select columns from the records. There are no data type modifications for this table.

8. On the command bar, click **Close and Apply**.

Check table relationships

1. Click on the Model tile on the left navigation pane and validate that Power BI shows relationships between all three tables.

2. Double-click each relationship and make sure that the **Cross-filter direction** is set to **Both**. This enables your visuals to refresh when a filter is applied.

Build a report

1. Click on the Report tile on the left navigation pane to explore data through visualizations. For text fields, tables and cards are useful visualizations.

2. Choose fields from each of the three tables to fill in the table or card.

Sample Power BI template - Azure portal only

When creating a knowledge store using the Azure portal, you have the option of downloading a Power BI template on the second page of the **Import data** wizard. This template gives you several visualizations, such as WordCloud and Network Navigator, for text-based content.

Click **Get Power BI Template** on the **Add cognitive skills** page to retrieve and download the template from its public GitHub location. The wizard modifies the template to accommodate the shape of your data, as captured in the knowledge store projections specified in the wizard. For this reason, the template you download will vary each time you run the wizard, assuming different data inputs and skill selections.

> **Note**
>
> The template is downloaded while the wizard is in mid-flight. You'll have to wait until the knowledge store is actually created in Azure Table Storage before you can use it.

Video introduction

For a demonstration of using Power BI with a knowledge store, watch the following video.

https://www.youtube-nocookie.com/embed/XWzLBP8iWqg?version=3

Next steps

Tables in Power BI reports and dashboards

Additional resources

Documentation

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Monitor queries - Azure Cognitive Search

Monitor query metrics for performance and throughput. Collect and analyze query string inputs in resource logs.

Azure Blob indexer - Azure Cognitive Search

Set up an Azure Blob indexer to automate indexing of blob content for full text search operations and knowledge mining in Azure Cognitive Search.

Document Extraction cognitive skill - Azure Cognitive Search

Extracts content from a file within the enrichment pipeline.

REST Tutorial: AI on Azure blobs - Azure Cognitive Search

Step through an example of text extraction and natural language processing over content in Blob Storage using Postman and the Azure Cognitive Search REST APIs.

Content metadata properties - Azure Cognitive Search

Metadata properties can provide content to fields in a search index. This article lists metadata properties supported in Azure Cognitive Search.

Data sources gallery - Azure Cognitive Search

Lists all of the supported data sources for importing into an Azure Cognitive Search index.

Custom Entity Lookup cognitive search skill - Azure Cognitive Search

Extract different custom entities from text in an Azure Cognitive Search cognitive search pipeline.

Show 5 more

Training

Learning certificate

Microsoft Certified: Power BI Data Analyst Associate - Certifications

Power BI data analysts design and build scalable data models, clean and transform data, and enable advanced analytic capabilities that provide meaningful business value.

Creating queries in Azure Cognitive Search

Article • 03/23/2023 • 6 minutes to read

If you're building a query for the first time, this article describes approaches and methods for setting up the request. It also introduces a query structure, and explains how field attributes and linguistic analyzers can impact query outcomes.

What's a query request?

A query is a read-only request against the docs collection of a single search index. It specifies a 'search' parameter, which contains the query expression consisting of terms, quote-enclosed phrases, and operators.

Other parameters on the request provide more definition to the query and response. For example, 'searchFields' scopes query execution to specific fields, 'select' specifies which fields are returned in results, and 'count' returns the number of matches found in the index.

The following example gives you a general idea of a query request by showing some of the available parameters. For more information about query composition, see Query types and compositions and Search Documents (REST).

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "NY +view",
    "queryType": "simple",
    "searchMode": "all",
    "searchFields": "HotelName, Description, Address/City, Address/StateProvince, Tags",
    "select": "HotelName, Description, Address/City, Address/StateProvince, Tags",
    "count": "true"
}
```

Choose a client

For early development and proof-of-concept testing, we recommend starting with an interactive tool like Azure portal, or the Postman app for making REST API calls. With

these approaches, you can test a query request in isolation and assess the effects of different properties without having to write any code.

To call search from within an app, we recommend the Azure.Document.Search client libraries in the Azure SDKs for .NET, Java, JavaScript, and Python.

Permissions

A query request requires read permissions, granted via an API key passed in the header. Any operation, including query requests, will work under an admin API key, but query requests can optionally use a query API key. Query API keys are strongly recommended. You can create up to 50 per service and assign different keys to different applications.

In Azure portal, access to the built-in tools, wizards, and objects require membership in the Contributor role or higher on the search service.

Use Azure portal to query an index

Search explorer (portal) is a query interface in the Azure portal that runs queries against indexes on the underlying search service. Internally, the portal makes Search Documents requests, but can't invoke Autocomplete, Suggestions, or Document Lookup.

You can select any index and REST API version, including preview. A query string can use simple or full syntax, with support for all query parameters (filter, select, searchFields, and so on). In the portal, when you open an index, you can work with Search Explorer alongside the index JSON definition in side-by-side tabs for easy access to field attributes. Check what fields are searchable, sortable, filterable, and facetable while testing queries.

Use a REST client

The Postman app can function as a query client. Using the app, you can connect to your search service and send Search Documents (REST) requests. Numerous tutorials and examples demonstrate REST clients for querying indexing.

Start with Create a search index using REST and Postman for step-by-step instructions for setting up requests.

Each request is standalone, so you must provide the endpoint, index name, and API version on every request. Other properties, Content-Type and API key, are passed on the request header. For more information, see Search Documents (REST) for help with formulating query requests.

Use an Azure SDK

For Cognitive Search, the Azure SDKs implement generally available features. As such, you can use any of the SDKs to query an index. All of them provide a **SearchClient** that has methods to interacting with an index, from loading an index with search documents, to formulating query requests.

Azure SDK	Client	Examples
.NET	SearchClient	DotNetHowTo
Java	SearchClient	SearchForDynamicDocumentsExample.java
JavaScript	SearchClient	Pending.
Python	SearchClient	sample_simple_query.py

Choose a query type: simple | full

If your query is full text search, a query parser is used to process any text that's passed as search terms and phrases. Azure Cognitive Search offers two query parsers.

- The simple parser understands the simple query syntax. This parser was selected as the default for its speed and effectiveness in free form text queries. The syntax supports common search operators (AND, OR, NOT) for term and phrase searches, and prefix (`*`) search (as in "sea*" for Seattle and Seaside). A general recommendation is to try the simple parser first, and then move on to full parser if application requirements call for more powerful queries.
- The full Lucene query syntax, enabled when you add `queryType=full` to the request, is based on the Apache Lucene Parser .

Full syntax and simple syntax overlap to the extent that both support the same prefix and boolean operations, but the full syntax provides more operators. In full, there are more operators for boolean expressions, and more operators for advanced queries such as fuzzy search, wildcard search, proximity search, and regular expressions.

Choose query methods

Search is fundamentally a user-driven exercise, where terms or phrases are collected from a search box, or from click events on a page. The following table summarizes the mechanisms by which you can collect user input, along with the expected search experience.

Input	Experience
Search method	A user types the terms or phrases into a search box, with or without operators, and clicks Search to send the request. Search can be used with filters on the same request, but not with autocomplete or suggestions.
Autocomplete method	A user types a few characters, and queries are initiated after each new character is typed. The response is a completed string from the index. If the string provided is valid, the user clicks Search to send that query to the service.
Suggestions method	As with autocomplete, a user types a few characters and incremental queries are generated. The response is a dropdown list of matching documents, typically represented by a few unique or descriptive fields. If any of the selections are valid, the user clicks one and the matching document is returned.
Faceted navigation	A page shows clickable navigation links or breadcrumbs that narrow the scope of the search. A faceted navigation structure is composed dynamically based on an initial query. For example, `search=*` to populate a faceted navigation tree composed of every possible category. A faceted navigation structure is created from a query response, but it's also a mechanism for expressing the next query. n REST API reference, `facets` is documented as a query parameter of a Search Documents operation, but it can be used without the `search` parameter.
Filter method	Filters are used with facets to narrow results. You can also implement a filter behind the page, for example to initialize the page with language-specific fields. In REST API reference, `$filter` is documented as a query parameter of a Search Documents operation, but it can be used without the `search` parameter.

Effect of field attributes on queries

If you're familiar with query types and composition, you might remember that the parameters on a query request depend on field attributes in an index. For example, only fields marked as *searchable* and *retrievable* can be used in queries and search results. When setting the `search`, `filter`, and `orderby` parameters in your request, you should check attributes to avoid unexpected results.

In the portal screenshot below of the hotels sample index, only the last two fields "LastRenovationDate" and "Rating" can be used in an `"$orderby"` only clause.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACETABLE	SEARCHABLE
HotelId	Edm.String	✓	✓		✓	
HotelName	Edm.String	✓				✓
Description	Edm.String	✓				✓
Description_fr	Edm.String	✓				✓
Category	Edm.String	✓	✓		✓	✓
Tags	Collection(E...	✓	✓		✓	✓
ParkingIncluded	Edm.Boolean	✓	✓		✓	
LastRenovationDate	Edm.DateTi...	✓		✓		
Rating	Edm.Double	✓	✓	✓	✓	

For field attribute definitions, see Create Index (REST API).

Effect of tokens on queries

During indexing, the search engine uses a text analyzer on strings to maximize the potential for finding a match at query time. At a minimum, strings are lower-cased, but depending on the analyzer, might also undergo lemmatization and stop word removal. Larger strings or compound words are typically broken up by whitespace, hyphens, or dashes, and indexed as separate tokens.

The point to take away here's that what you think your index contains, and what's actually in it, can be different. If queries don't return expected results, you can inspect the tokens created by the analyzer through the Analyze Text (REST API). For more information about tokenization and the impact on queries, see Partial term search and patterns with special characters.

Next steps

Now that you have a better understanding of how query requests work, try the following quickstarts for hands-on experience.

- Search explorer
- How to query in REST
- How to query in .NET
- How to query in Python
- How to query in JavaScript

Use the "simple" search syntax in Azure Cognitive Search

Article • 11/29/2022 • 12 minutes to read

In Azure Cognitive Search, the simple query syntax invokes the default query parser for full text search. The parser is fast and handles common scenarios, including full text search, filtered and faceted search, and prefix search. This article uses examples to illustrate simple syntax usage in a Search Documents (REST API) request.

> **Note**
>
> An alternative query syntax is **Full Lucene**, supporting more complex query structures, such as fuzzy and wildcard search. For more information and examples, see **Use the full Lucene syntax**.

Hotels sample index

The following queries are based on the hotels-sample-index, which you can create by following the instructions in this quickstart.

Example queries are articulated using the REST API and POST requests. You can paste and run them in Postman or another web client.

Request headers must have the following values:

Key	Value
Content-Type	application/json
api-key	`<your-search-service-api-key>`, either query or admin key

URI parameters must include your search service endpoint with the index name, docs collections, search command, and API version, similar to the following example:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
```

Request body should be formed as valid JSON:

JSON

```
{
    "search": "*",
    "queryType": "simple",
    "select": "HotelId, HotelName, Category, Tags, Description",
    "count": true
}
```

- "search" set to `*` is an unspecified query, equivalent to null or empty search. It's not especially useful, but it is the simplest search you can do, and it shows all retrievable fields in the index, with all values.
- "queryType" set to "simple" is the default and can be omitted, but it's included to further reinforce that the query examples in this article are expressed in the simple syntax.
- "select" set to a comma-delimited list of fields is used for search result composition, including just those fields that are useful in the context of search results.
- "count" returns the number of documents matching the search criteria. On an empty search string, the count will be all documents in the index (50 in the case of hotels-sample-index).

Example 1: Full text search

Full text search can be any number of standalone terms or quote-enclosed phrases, with or without boolean operators.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "pool spa +airport",
    "searchMode": "any",
    "queryType": "simple",
    "select": "HotelId, HotelName, Category, Description",
    "count": true
}
```

A keyword search that's composed of important terms or phrases tend to work best. String fields undergo text analysis during indexing and querying, dropping non-essential words like "the", "and", "it". To see how a query string is tokenized in the index, pass the string in an Analyze Text call to the index.

The "searchMode" parameter controls precision and recall. If you want more recall, use the default "any" value, which returns a result if any part of the query string is matched. If you favor precision, where all parts of the string must be matched, change searchMode to "all". Try the above query both ways to see how searchMode changes the outcome.

Response for the "pool spa +airport" query should look similar to the following example, trimmed for brevity.

JSON


```
"@odata.count": 6,
"value": [
    {
        "@search.score": 7.3617697,
        "HotelId": "21",
        "HotelName": "Nova Hotel & Spa",
        "Description": "1 Mile from the airport.  Free WiFi, Outdoor Pool,
Complimentary Airport Shuttle, 6 miles from the beach & 10 miles from
downtown.",
        "Category": "Resort and Spa",
        "Tags": [
            "pool",
            "continental breakfast",
            "free parking"
        ]
    },
    {
        "@search.score": 2.5560288,
        "HotelId": "25",
        "HotelName": "Scottish Inn",
        "Description": "Newly Redesigned Rooms & airport shuttle.  Minutes
from the airport, enjoy lakeside amenities, a resort-style pool & stylish
new guestrooms with Internet TVs.",
        "Category": "Luxury",
        "Tags": [
            "24-hour front desk service",
            "continental breakfast",
            "free wifi"
        ]
    },
    {
        "@search.score": 2.2988036,
        "HotelId": "35",
        "HotelName": "Suites At Bellevue Square",
        "Description": "Luxury at the mall.  Located across the street from
the Light Rail to downtown.  Free shuttle to the mall and airport.",
        "Category": "Resort and Spa",
        "Tags": [
            "continental breakfast",
            "air conditioning",
            "24-hour front desk service"
```

```
        ]
    }
```

Notice the search score in the response. This is the relevance score of the match. By default, a search service will return the top 50 matches based on this score.

Uniform scores of "1.0" occur when there is no rank, either because the search was not full text search, or because no criteria was provided. For example, in an empty search (search=`*`), rows come back in arbitrary order. When you include actual criteria, you will see search scores evolve into meaningful values.

Example 2: Look up by ID

When you return search results in a query, a logical next step is to provide a details page that includes more fields from the document. This example shows you how to return a single document using Lookup Document by passing in the document ID.

HTTP

```
GET /indexes/hotels-sample-index/docs/41?api-version=2020-06-30
```

All documents have a unique identifier. If you are using the portal, select the index from **Indexes** tab and then look at the field definitions to determine which field is the key. Using REST, the Get Index call returns the index definition in the response body.

Response for the above query consists of the document whose key is 41. Any field that is marked as "retrievable" in the index definition can be returned in search results and rendered in your app.

JSON

```
{
    "HotelId": "41",
    "HotelName": "Ocean Air Motel",
    "Description": "Oceanfront hotel overlooking the beach features rooms
with a private balcony and 2 indoor and outdoor pools. Various shops and art
entertainment are on the boardwalk, just steps away.",
    "Description_fr": "L'hôtel front de mer surplombant la plage dispose de
chambres avec balcon privé et 2 piscines intérieures et extérieures. Divers
commerces et animations artistiques sont sur la promenade, à quelques pas.",
    "Category": "Budget",
    "Tags": [
        "pool",
        "air conditioning",
        "bar"
    ],
```

```
    "ParkingIncluded": true,
    "LastRenovationDate": "1951-05-10T00:00:00Z",
    "Rating": 3.5,
    "Location": {
        "type": "Point",
        "coordinates": [
            -157.846817,
            21.295841
        ],
        "crs": {
            "type": "name",
            "properties": {
                "name": "EPSG:4326"
            }
        }
    },
    "Address": {
        "StreetAddress": "1450 Ala Moana Blvd 2238 Ala Moana Ctr",
        "City": "Honolulu",
        "StateProvince": "HI",
        "PostalCode": "96814",
        "Country": "USA"
    },
```

Example 3: Filter on text

Filter syntax is an OData expression that you can use by itself or with "search". Used together, "filter" is applied first to the entire index, and then the search is performed on the results of the filter. Filters can therefore be a useful technique to improve query performance since they reduce the set of documents that the search query needs to process.

Filters can be defined on any field marked as "filterable" in the index definition. For hotels-sample-index, filterable fields include Category, Tags, ParkingIncluded, Rating, and most Address fields.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "art tours",
    "queryType": "simple",
    "filter": "Category eq 'Resort and Spa'",
    "select": "HotelId,HotelName,Description,Category",
    "count": true
}
```

Response for the above query is scoped to only those hotels categorized as "Report and Spa", that include the terms "art" or "tours". In this case, there is just one match.

JSON

```
{
    "@search.score": 2.8576312,
    "HotelId": "31",
    "HotelName": "Santa Fe Stay",
    "Description": "Nestled on six beautifully landscaped acres, located 2
blocks from the Plaza. Unwind at the spa and indulge in art tours on site.",
    "Category": "Resort and Spa"
}
```

Example 4: Filter functions

Filter expressions can include "search.ismatch" and "search.ismatchscoring" functions, allowing you to build a search query within the filter. This filter expression uses a wildcard on *free* to select amenities including free wifi, free parking, and so forth.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
  {
    "search": "",
    "filter": "search.ismatch('free*', 'Tags', 'full', 'any')",
    "select": "HotelId, HotelName, Category, Description",
    "count": true
  }
```

Response for the above query matches on 19 hotels that offer free amenities. Notice that the search score is a uniform "1.0" throughout the results. This is because the search expression is null or empty, resulting in verbatim filter matches, but no full text search. Relevance scores are only returned on full text search. If you are using filters without "search", make sure you have sufficient sortable fields so that you can control search rank.

JSON

```
"@odata.count": 19,
"value": [
    {
        "@search.score": 1.0,
        "HotelId": "31",
        "HotelName": "Santa Fe Stay",
        "Tags": [
            "view",
```

```
            "restaurant",
            "free parking"
        ]
    },
    {
        "@search.score": 1.0,
        "HotelId": "27",
        "HotelName": "Super Deluxe Inn & Suites",
        "Tags": [
            "bar",
            "free wifi"
        ]
    },
    {
        "@search.score": 1.0,
        "HotelId": "39",
        "HotelName": "Whitefish Lodge & Suites",
        "Tags": [
            "continental breakfast",
            "free parking",
            "free wifi"
        ]
    },
    {
        "@search.score": 1.0,
        "HotelId": "11",
        "HotelName": "Regal Orb Resort & Spa",
        "Tags": [
            "free wifi",
            "restaurant",
            "24-hour front desk service"
        ]
    },
```

Example 5: Range filters

Range filtering is supported through filters expressions for any data type. The following examples illustrate numeric and string ranges. Data types are important in range filters and work best when numeric data is in numeric fields, and string data in string fields. Numeric data in string fields is not suitable for ranges because numeric strings are not comparable.

The following query is a numeric range. In hotels-sample-index, the only filterable numeric field is Rating.

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "*",
```

```
    "filter": "Rating ge 2 and Rating lt 4",
    "select": "HotelId, HotelName, Rating",
    "orderby": "Rating desc",
    "count": true
}
```

Response for this query should look similar to the following example, trimmed for brevity.

JSON

```
"@odata.count": 27,
"value": [
    {
        "@search.score": 1.0,
        "HotelId": "22",
        "HotelName": "Stone Lion Inn",
        "Rating": 3.9
    },
    {
        "@search.score": 1.0,
        "HotelId": "25",
        "HotelName": "Scottish Inn",
        "Rating": 3.8
    },
    {
        "@search.score": 1.0,
        "HotelId": "2",
        "HotelName": "Twin Dome Motel",
        "Rating": 3.6
    }
```

The next query is a range filter over a string field (Address/StateProvince):

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "*",
    "filter": "Address/StateProvince ge 'A*' and Address/StateProvince lt
'D*'",
    "select": "HotelId, HotelName, Address/StateProvince",
    "count": true
}
```

Response for this query should look similar to the example below, trimmed for brevity. In this example, it's not possible to sort by StateProvince because the field is not attributed as "sortable" in the index definition.

JSON

```
"@odata.count": 9,
"value": [
    {
        "@search.score": 1.0,
        "HotelId": "9",
        "HotelName": "Smile Hotel",
        "Address": {
            "StateProvince": "CA "
        }
    },
    {
        "@search.score": 1.0,
        "HotelId": "39",
        "HotelName": "Whitefish Lodge & Suites",
        "Address": {
            "StateProvince": "CO"
        }
    },
    {
        "@search.score": 1.0,
        "HotelId": "7",
        "HotelName": "Countryside Resort",
        "Address": {
            "StateProvince": "CA "
        }
    },
```

Example 6: Geospatial search

The hotels-sample index includes a Location field with latitude and longitude coordinates. This example uses the geo.distance function that filters on documents within the circumference of a starting point, out to an arbitrary distance (in kilometers) that you provide. You can adjust the last value in the query (10) to reduce or enlarge the surface area of the query.

HTTP

```
POST /indexes/v/docs/search?api-version=2020-06-30
{
    "search": "*",
    "filter": "geo.distance(Location, geography'POINT(-122.335114
47.612839)') le 10",
    "select": "HotelId, HotelName, Address/City, Address/StateProvince",
    "count": true
}
```

Response for this query returns all hotels within a 10 kilometer distance of the coordinates provided:

JSON

```
{
    "@odata.count": 3,
    "value": [
        {
            "@search.score": 1.0,
            "HotelId": "45",
            "HotelName": "Arcadia Resort & Restaurant",
            "Address": {
                "City": "Seattle",
                "StateProvince": "WA"
            }
        },
        {
            "@search.score": 1.0,
            "HotelId": "24",
            "HotelName": "Gacc Capital",
            "Address": {
                "City": "Seattle",
                "StateProvince": "WA"
            }
        },
        {
            "@search.score": 1.0,
            "HotelId": "16",
            "HotelName": "Double Sanctuary Resort",
            "Address": {
                "City": "Seattle",
                "StateProvince": "WA"
            }
        }
    ]
}
```

Example 7: Booleans with searchMode

Simple syntax supports boolean operators in the form of characters (`+, -, |`) to support AND, OR, and NOT query logic. Boolean search behaves as you might expect, with a few noteworthy exceptions.

In previous examples, the "searchMode" parameter was introduced as a mechanism for influencing precision and recall, with "searchMode=any" favoring recall (a document that satisfies any of the criteria is considered a match), and "searchMode=all" favoring precision (all criteria must be matched in a document).

In the context of a Boolean search, the default "searchMode=any" can be confusing if you are stacking a query with multiple operators and getting broader instead of narrower results. This is particularly true with NOT, where results include all documents "not containing" a specific term or phrase.

The following example provides an illustration. Running the following query with searchMode (any), 42 documents are returned: those containing the term "restaurant", plus all documents that do not have the phrase "air conditioning".

Notice that there is no space between the boolean operator (`-`) and the phrase "air conditioning".

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "restaurant -\"air conditioning\"",
    "searchMode": "any",
    "searchFields": "Tags",
    "select": "HotelId, HotelName, Tags",
    "count": true
}
```

Changing to "searchMode=all" enforces a cumulative effect on criteria and returns a smaller result set (7 matches) consisting of documents containing the term "restaurant", minus those containing the phrase "air conditioning".

Response for this query would now look similar to the following example, trimmed for brevity.

JSON

```
"@odata.count": 7,
"value": [
    {
        "@search.score": 2.5460577,
        "HotelId": "11",
        "HotelName": "Regal Orb Resort & Spa",
        "Tags": [
            "free wifi",
            "restaurant",
            "24-hour front desk service"
        ]
    },
    {
        "@search.score": 2.166792,
        "HotelId": "10",
        "HotelName": "Countryside Hotel",
        "Tags": [
```

```
            "24-hour front desk service",
            "coffee in lobby",
            "restaurant"
        ]
    },
```

Example 8: Paging results

In previous examples, you learned about parameters that affect search results composition, including "select" that determines which fields are in a result, sort orders, and how to include a count of all matches. This example is a continuation of search result composition in the form of paging parameters that allow you to batch the number of results that appear in any given page.

By default, a search service returns the top 50 matches. To control the number of matches in each page, use "top" to define the size of the batch, and then use "skip" to pick up subsequent batches.

The following example uses a filter and sort order on the Rating field (Rating is both filterable and sortable) because it's easier to see the effects of paging on sorted results. In a regular full search query, the top matches are ranked and paged by "@search.score".

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "*",
    "filter": "Rating gt 4",
    "select": "HotelName, Rating",
    "orderby": "Rating desc",
    "top": "5",
    "count": true
}
```

The query finds 21 matching documents, but because you specified "top", the response returns just the top five matches, with ratings starting at 4.9, and ending at 4.7 with "Lady of the Lake B & B".

To get the next 5, skip the first batch:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "*",
```

```
    "filter": "Rating gt 4",
    "select": "HotelName, Rating",
    "orderby": "Rating desc",
    "top": "5",
    "skip": "5",
    "count": true
}
```

The response for the second batch skips the first five matches, returning the next five, starting with "Pull'r Inn Motel". To continue with additional batches, you would keep "top" at 5, and then increment "skip" by 5 on each new request (skip=5, skip=10, skip=15, and so forth).

JSON

```
"value": [
    {
        "@search.score": 1.0,
        "HotelName": "Pull'r Inn Motel",
        "Rating": 4.7
    },
    {
        "@search.score": 1.0,
        "HotelName": "Sublime Cliff Hotel",
        "Rating": 4.6
    },
    {
        "@search.score": 1.0,
        "HotelName": "Antiquity Hotel",
        "Rating": 4.5
    },
    {
        "@search.score": 1.0,
        "HotelName": "Nordick's Motel",
        "Rating": 4.5
    },
    {
        "@search.score": 1.0,
        "HotelName": "Winter Panorama Resort",
        "Rating": 4.5
    }
]
```

Next steps

Now that you have some practice with the basic query syntax, try specifying queries in code. The following links explain how to set up search queries using the Azure SDKs.

- Query your index using the .NET SDK

- Query your index using the Python SDK
- Query your index using the JavaScript SDK

Additional syntax reference, query architecture, and examples can be found in the following links:

- Lucene syntax query examples for building advanced queries
- How full text search works in Azure Cognitive Search
- Simple query syntax
- Full Lucene query syntax
- Filter syntax

Add spell check to queries in Cognitive Search

Article • 03/28/2023 • 4 minutes to read

> **Important**
>
> Spell correction is in public preview under **supplemental terms of use**. It's available through the Azure portal and preview REST API only.

You can improve recall by spell-correcting individual search query terms before they reach the search engine. The **speller** parameter is supported for all query types: simple, full, and the semantic option currently in public preview.

Speller was released in tandem with the semantic search preview and shares the "queryLanguage" parameter, but is otherwise an independent feature with its own prerequisites. There's no sign-up or extra charges for using this feature.

Prerequisites

To use spell check, you'll need the following:

- A search service at Basic tier or above, in any region.
- An existing search index with content in a supported language.
- A query request that has "speller=lexicon", and "queryLanguage" set to a supported language. Spell check works on strings passed in the "search" parameter. It's not supported for filters.

Use a search client that supports preview APIs on the query request. For REST, you can use Postman, another web client, or code that you've modified to make REST calls to the preview APIs. You can also use beta releases of the Azure SDKs.

Client library	Versions
REST API	2021-04-30-Preview or 2020-06-30-Preview
Azure SDK for .NET	version 11.5.0-beta.2
Azure SDK for Java	version 11.6.0-beta.5
Azure SDK for JavaScript	version 11.3.0-beta.8

Client library	Versions
Azure SDK for Python	version 11.4.0b3

Spell correction with simple search

The following example uses the built-in hotels-sample index to demonstrate spell correction on a simple free form text query. Without spell correction, the query returns zero results. With correction, the query returns one result for Johnson's family-oriented resort.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30-Preview
{
    "search": "famly acitvites",
    "speller": "lexicon",
    "queryLanguage": "en-us",
    "queryType": "simple",
    "select": "HotelId,HotelName,Description,Category,Tags",
    "count": true
}
```

Spell correction with full Lucene

Spelling correction occurs on individual query terms that undergo text analysis, which is why you can use the speller parameter with some Lucene queries, but not others.

- Incompatible query forms that bypass text analysis include: wildcard, regex, fuzzy
- Compatible query forms include: fielded search, proximity, term boosting

This example uses fielded search over the Category field, with full Lucene syntax, and a misspelled query term. By including speller, the typo in "Suiite" is corrected and the query succeeds.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30-Preview
{
    "search": "Category:(Resort and Spa) OR Category:Suiite",
    "queryType": "full",
    "speller": "lexicon",
    "queryLanguage": "en-us",
```

```
    "select": "Category",
    "count": true
}
```

Spell correction with semantic search

This query, with typos in every term except one, undergoes spelling corrections to return relevant results. To learn more, see Configure semantic ranking.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30-Preview
{
    "search": "hisotoric hotell wiht great restrant nad wiifi",
    "queryType": "semantic",
    "speller": "lexicon",
    "queryLanguage": "en-us",
    "searchFields": "HotelName,Tags,Description",
    "select": "HotelId,HotelName,Description,Category,Tags",
    "count": true
}
```

Supported languages

Valid values for queryLanguage can be found in the following table, copied from the list of supported languages (REST API reference).

Language	queryLanguage
English [EN]	EN, EN-US (default)
Spanish [ES]	ES, ES-ES (default)
French [FR]	FR, FR-FR (default)
German [DE]	DE, DE-DE (default)
Dutch [NL]	NL, NL-BE, NL-NL (default)

queryLanguage considerations

As noted elsewhere, a query request can only have one queryLanguage parameter, but that parameter is shared by multiple features, each of which supports a different cohort

of languages. If you're just using spell check, the list of supported languages in the above table is the complete list.

Language analyzer considerations

Indexes that contain non-English content often use language analyzers on non-English fields to apply the linguistic rules of the native language.

If you're now adding spell check to content that also undergoes language analysis, you'll achieve better results if you use the same language at every step of indexing and query processing. For example, if a field's content was indexed using the "fr.microsoft" language analyzer, then queries, spell check, semantic captions, and semantic answers should all use a French lexicon or language library of some form.

To recap how language libraries are used in Cognitive Search:

- Language analyzers can be invoked during indexing and query execution, and are either Apache Lucene (for example, "de.lucene") or Microsoft ("de.microsoft).
- Language lexicons invoked during spell check are specified using one of the language codes in the table above.

In a query request, the value assigned to queryLanguage applies equally to speller, answers, and captions.

> **Note**
>
> Language consistency across various property values is only a concern if you are using language analyzers. If you are using language-agnostic analyzers (such as keyword, simple, standard, stop, whitespace, or `standardasciifolding.lucene`), then the queryLanguage value can be whatever you want.

While content in a search index can be composed in multiple languages, the query input is most likely in one. The search engine doesn't check for compatibility of queryLanguage, language analyzer, and the language in which content is composed, so be sure to scope queries accordingly to avoid producing incorrect results.

Next steps

- Invoke semantic ranking and captions
- Create a basic query
- Use full Lucene query syntax

- Use simple query syntax
- Semantic search overview

Create a suggester to enable autocomplete and suggested results in a query

Article • 03/10/2023 • 6 minutes to read

In Azure Cognitive Search, typeahead or "search-as-you-type" is enabled through a *suggester*. A suggester provides a list of fields that undergo extra tokenization, generating prefix sequences to support matches on partial terms. For example, a suggester that includes a City field with a value for "Seattle" will have prefix combinations of "sea", "seat", "seatt", and "seattl" to support typeahead.

Matches on partial terms can be either an autocompleted query or a suggested match. The same suggester supports both experiences.

Typeahead experiences in Cognitive Search

Typeahead can be *autocomplete*, which completes a partial input for a whole term query, or *suggestions* that invite click through to a particular match. Autocomplete produces a query. Suggestions produce a matching document.

The following screenshot from Create your first app in C# illustrates both. Autocomplete anticipates a potential term, finishing "tw" with "in". Suggestions are mini search results, where a field like hotel name represents a matching hotel search document from the index. For suggestions, you can surface any field that provides descriptive information.

You can use these features separately or together. To implement these behaviors in Azure Cognitive Search, there's an index and query component.

- Add a suggester to a search index definition. The remainder of this article is focused on creating a suggester.

- Call a suggester-enabled query, in the form of a Suggestion request or Autocomplete request, using one of the APIs listed below.

Search-as-you-type support is enabled on a per-field basis for string fields. You can implement both typeahead behaviors within the same search solution if you want an experience similar to the one indicated in the screenshot. Both requests target the *documents* collection of specific index and responses are returned after a user has provided at least a three character input string.

How to create a suggester

To create a suggester, add one to an index definition. A suggester takes a name and a collection of fields over which the typeahead experience is enabled. The best time to create a suggester is when you're also defining the field that will use it.

- Use string fields only.
- If the string field is part of a complex type (for example, a City field within Address), include the parent in the field path: `"Address/City"` (REST and C# and Python), or `["Address"]["City"]` (JavaScript).
- Use the default standard Lucene analyzer (`"analyzer": null`) or a language analyzer (for example, `"analyzer": "en.Microsoft"`) on the field.

If you try to create a suggester using pre-existing fields, the API will disallow it. Prefixes are generated during indexing, when partial terms in two or more character combinations are tokenized alongside whole terms. Given that existing fields are already tokenized, you'll have to rebuild the index if you want to add them to a suggester. For more information, see How to rebuild an Azure Cognitive Search index.

Choose fields

Although a suggester has several properties, it's primarily a collection of string fields for which you're enabling a search-as-you-type experience. There's one suggester for each index, so the suggester list must include all fields that contribute content for both suggestions and autocomplete.

Autocomplete benefits from a larger pool of fields to draw from because the additional content has more term completion potential.

Suggestions, on the other hand, produce better results when your field choice is selective. Remember that the suggestion is a proxy for a search document so you'll want

fields that best represent a single result. Names, titles, or other unique fields that distinguish among multiple matches work best. If fields consist of repetitive values, the suggestions consist of identical results and a user won't know which one to click.

To satisfy both search-as-you-type experiences, add all of the fields that you need for autocomplete, but then use "$select", "$top", "$filter", and "searchFields" to control results for suggestions.

Choose analyzers

Your choice of an analyzer determines how fields are tokenized and subsequently prefixed. For example, for a hyphenated string like "context-sensitive", using a language analyzer will result in these token combinations: "context", "sensitive", "context-sensitive". Had you used the standard Lucene analyzer, the hyphenated string wouldn't exist.

When evaluating analyzers, consider using the Analyze Text API for insight into how terms are processed. Once you build an index, you can try various analyzers on a string to view token output.

Fields that use custom analyzers or built-in analyzers (with the exception of standard Lucene) are explicitly disallowed to prevent poor outcomes.

> **Note**
>
> If you need to work around the analyzer constraint, for example if you need a keyword or ngram analyzer for certain query scenarios, you should use two separate fields for the same content. This will allow one of the fields to have a suggester, while the other can be set up with a custom analyzer configuration.

Create using the portal

When using **Add Index** or the **Import data** wizard to create an index, you have the option of enabling a suggester:

1. In the index definition, enter a name for the suggester.
2. In each field definition for new fields, select a checkbox in the Suggester column. A checkbox is available on string fields only.

As previously noted, analyzer choice impacts tokenization and prefixing. Consider the entire field definition when enabling suggesters.

Create using REST

In the REST API, add suggesters through Create Index or Update Index.

JSON

```
{
  "name": "hotels-sample-index",
  "fields": [
    . . .
        {
            "name": "HotelName",
            "type": "Edm.String",
            "facetable": false,
            "filterable": false,
            "key": false,
            "retrievable": true,
            "searchable": true,
            "sortable": false,
            "analyzer": "en.microsoft",
            "indexAnalyzer": null,
            "searchAnalyzer": null,
            "synonymMaps": [],
            "fields": []
        },
  ],
  "suggesters": [
    {
      "name": "sg",
      "searchMode": "analyzingInfixMatching",
      "sourceFields": ["HotelName"]
    }
  ],
  "scoringProfiles": [
    . . .
  ]
}
```

Create using .NET

In C#, define a SearchSuggester object. `Suggesters` is a collection on a SearchIndex object, but it can only take one item. Add a suggester to the index definition.

C#

```
private static void CreateIndex(string indexName, SearchIndexClient
indexClient)
{
    FieldBuilder fieldBuilder = new FieldBuilder();
```

```
    var searchFields = fieldBuilder.Build(typeof(Hotel));

    var definition = new SearchIndex(indexName, searchFields);

    var suggester = new SearchSuggester("sg", new[] { "HotelName",
"Category", "Address/City", "Address/StateProvince" });
    definition.Suggesters.Add(suggester);

    indexClient.CreateOrUpdateIndex(definition);
}
```

Property reference

Property	Description
name	Specified in the suggester definition, but also called on an Autocomplete or Suggestions request.
sourceFields	Specified in the suggester definition. It's a list of one or more fields in the index that are the source of the content for suggestions. Fields must be of type `Edm.String`. If an analyzer is specified on the field, it must be a named lexical analyzer from this list (not a custom analyzer). As a best practice, specify only those fields that lend themselves to an expected and appropriate response, whether it's a completed string in a search bar or a dropdown list. A hotel name is a good candidate because it has precision. Verbose fields like descriptions and comments are too dense. Similarly, repetitive fields, such as categories and tags, are less effective. In the examples, we include "category" anyway to demonstrate that you can include multiple fields.
searchMode	REST-only parameter, but also visible in the portal. This parameter isn't available in the .NET SDK. It indicates the strategy used to search for candidate phrases. The only mode currently supported is `analyzingInfixMatching`, which currently matches on the beginning of a term.

Use a suggester

A suggester is used in a query. After a suggester is created, call one of the following APIs for a search-as-you-type experience:

- Suggestions REST API
- Autocomplete REST API
- SuggestAsync method
- AutocompleteAsync method

In a search application, client code should use a library like jQuery UI Autocomplete to collect the partial query and provide the match. For more information about this task, see Add autocomplete or suggested results to client code.

API usage is illustrated in the following call to the Autocomplete REST API. There are two takeaways from this example. First, as with all queries, the operation is against the documents collection of an index and the query includes a "search" parameter, which in this case provides the partial query. Second, you must add "suggesterName" to the request. If a suggester isn't defined in the index, a call to autocomplete or suggestions will fail.

HTTP

```
POST /indexes/myxboxgames/docs/autocomplete?search&api-version=2020-06-30
{
  "search": "minecraf",
  "suggesterName": "sg"
}
```

Sample code

- Add search to a web site (JavaScript) uses an open source Suggestions package for partial term completion in the client app.

Next steps

Learn more about requests\ formulation.

Add autocomplete and suggestions to client code

Additional resources

Documentation

Tutorial: create a custom analyzer - Azure Cognitive Search

Learn how to build a custom analyzer to improve the quality of search results in Azure Cognitive Search.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Load a search index - Azure Cognitive Search

Import and refresh data in a search index using the portal, REST APIs, or an Azure SDK.

Add language analyzers to string fields - Azure Cognitive Search

Multi-lingual lexical analysis for non-English queries and indexes in Azure Cognitive Search.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Add custom analyzers to string fields - Azure Cognitive Search

Configure text tokenizers and character filters to perform text analysis on strings during indexing and queries.

Show 5 more

Add autocomplete and suggestions to client apps using Azure Cognitive Search

Article • 03/10/2023 • 9 minutes to read

Search-as-you-type is a common technique for improving query productivity. In Azure Cognitive Search, this experience is supported through *autocomplete*, which finishes a term or phrase based on partial input (completing "micro" with "microsoft"). A second user experience is *suggestions*, or a short list of matching documents (returning book titles with an ID so that you can link to a detail page about that book). Both autocomplete and suggestions are predicated on a match in the index. The service won't offer queries that return zero results.

To implement these experiences in Azure Cognitive Search, you will need:

- A *suggester* definition that's embedded in the index schema.
- A *query* specifying Autocomplete or Suggestions API on the request.
- A *UI control* to handle search-as-you-type interactions in your client app. We recommend using an existing JavaScript library for this purpose.

In Azure Cognitive Search, autocompleted queries and suggested results are retrieved from the search index, from selected fields that you have registered with a suggester. A suggester is part of the index, and it specifies which fields will provide content that either completes a query, suggests a result, or does both. When the index is created and loaded, a suggester data structure is created internally to store prefixes used for matching on partial queries. For suggestions, choosing suitable fields that are unique, or at least not repetitive, is essential to the experience. For more information, see Create a suggester.

The remainder of this article is focused on queries and client code. It uses JavaScript and C# to illustrate key points. REST API examples are used to concisely present each operation. For links to end-to-end code samples, see Next steps.

Set up a request

Elements of a request include one of the search-as-you-type APIs, a partial query, and a suggester. The following script illustrates components of a request, using the Autocomplete REST API as an example.

```HTTP
POST /indexes/myxboxgames/docs/autocomplete?search&api-version=2020-06-30
{
  "search": "minecraf",
  "suggesterName": "sg"
}
```

The "suggesterName" gives you the suggester-aware fields used to complete terms or suggestions. For suggestions in particular, the field list should be composed of those that offer clear choices among matching results. On a site that sells computer games, the field might be the game title.

The "search" parameter provides the partial query, where characters are fed to the query request through the jQuery Autocomplete control. In the above example, "minecraf" is a static illustration of what the control might have passed in.

The APIs do not impose minimum length requirements on the partial query; it can be as little as one character. However, jQuery Autocomplete provides a minimum length. A minimum of two or three characters is typical.

Matches are on the beginning of a term anywhere in the input string. Given "the quick brown fox", both autocomplete and suggestions will match on partial versions of "the", "quick", "brown", or "fox" but not on partial infix terms like "rown" or "ox". Furthermore, each match sets the scope for downstream expansions. A partial query of "quick br" will match on "quick brown" or "quick bread", but neither "brown" or "bread" by themselves would be match unless "quick" precedes them.

APIs for search-as-you-type

Follow these links for the REST and .NET SDK reference pages:

- Suggestions REST API
- Autocomplete REST API
- SuggestAsync method
- AutocompleteAsync method

Structure a response

Responses for autocomplete and suggestions are what you might expect for the pattern: Autocomplete returns a list of terms, Suggestions returns terms plus a document ID so that you can fetch the document (use the Lookup Document API to fetch the specific document for a detail page).

Responses are shaped by the parameters on the request:

- For Autocomplete, set the autocompleteMode to determine whether text completion occurs on one or two terms.
- For Suggestions, set $select to return fields containing unique or differentiating values, such as names and description. Avoid fields that contain duplicate values (such as a category or city).

The following additional parameters apply to both autocomplete and suggestions, but are perhaps more necessary for suggestions, especially when a suggester includes multiple fields.

Parameter	Usage
searchFields	Constrain the query to specific fields.
$filter	Apply match criteria on the result set (`$filter=Category eq 'ActionAdventure'`).
$top	Limit the results to a specific number (`$top=5`).

Add user interaction code

Auto-filling a query term or dropping down a list of matching links requires user interaction code, typically JavaScript, that can consume requests from external sources, such as autocomplete or suggestion queries against an Azure Search Cognitive index.

Although you could write this code natively, it's much easier to use functions from existing JavaScript library, such as one of the following.

- Autocomplete widget (jQuery UI) appears in the Suggestion code snippet. You can create a search box, and then reference it in a JavaScript function that uses the Autocomplete widget. Properties on the widget set the source (an autocomplete or suggestions function), minimum length of input characters before action is taken, and positioning.
- XDSoft Autocomplete plug-in appears in the Autocomplete code snippet.
- suggestions appears in the JavaScript tutorial and code sample.

Use these libraries in the client to create a search box supporting both suggestions and autocomplete. Inputs collected in the search box can then be paired with suggestions and autocomplete actions on the search service.

Suggestions

This section walks you through an implementation of suggested results, starting with the search box definition. It also shows how and script that invokes the first JavaScript autocomplete library referenced in this article.

Create a search box

Assuming the jQuery UI Autocomplete library and an MVC project in C#, you could define the search box using JavaScript in the **Index.cshtml** file. The library adds the search-as-you-type interaction to the search box by making asynchronous calls to the MVC controller to retrieve suggestions.

In **Index.cshtml** under the folder \Views\Home, a line to create a search box might be as follows:

HTML

```html
<input class="searchBox" type="text" id="searchbox1" placeholder="search">
```

This example is a simple input text box with a class for styling, an ID to be referenced by JavaScript, and placeholder text.

Within the same file, embed JavaScript that references the search box. The following function calls the Suggest API, which requests suggested matching documents based on partial term inputs:

JavaScript

```javascript
$(function () {
    $("#searchbox1").autocomplete({
        source: "/home/suggest?highlights=false&fuzzy=false&",
        minLength: 3,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        }
    });
});
```

The `source` tells the jQuery UI Autocomplete function where to get the list of items to show under the search box. Since this project is an MVC project, it calls the **Suggest** function in **HomeController.cs** that contains the logic for returning query suggestions.

This function also passes a few parameters to control highlights, fuzzy matching, and term. The autocomplete JavaScript API adds the term parameter.

The `minLength: 3` ensures that recommendations will only be shown when there are at least three characters in the search box.

Enable fuzzy matching

Fuzzy search allows you to get results based on close matches even if the user misspells a word in the search box. The edit distance is 1, which means there can be a maximum discrepancy of one character between the user input and a match.

JavaScript

```
source: "/home/suggest?highlights=false&fuzzy=true&",
```

Enable highlighting

Highlighting applies font style to the characters in the result that correspond to the input. For example, if the partial input is "micro", the result would appear as **micro**soft, **micro**scope, and so forth. Highlighting is based on the HighlightPreTag and HighlightPostTag parameters, defined inline with the Suggestion function.

JavaScript

```
source: "/home/suggest?highlights=true&fuzzy=true&",
```

Suggest function

If you are using C# and an MVC application, **HomeController.cs** file under the Controllers directory is where you might create a class for suggested results. In .NET, a Suggest function is based on the SuggestAsync method. For more information about the .NET SDK, see How to use Azure Cognitive Search from a .NET Application.

The `InitSearch` method creates an authenticated HTTP index client to the Azure Cognitive Search service. Properties on the SuggestOptions class determine which fields are searched and returned in the results, the number of matches, and whether fuzzy matching is used.

For autocomplete, fuzzy matching is limited to one edit distance (one omitted or misplaced character). Note that fuzzy matching in autocomplete queries can sometimes

produce unexpected results depending on index size and how it's sharded. For more information, see partition and sharding concepts.

C#

```
public async Task<ActionResult> SuggestAsync(bool highlights, bool fuzzy,
string term)
{
    InitSearch();

    var options = new SuggestOptions()
    {
        UseFuzzyMatching = fuzzy,
        Size = 8,
    };

    if (highlights)
    {
        options.HighlightPreTag = "<b>";
        options.HighlightPostTag = "</b>";
    }

    // Only one suggester can be specified per index.
    // The suggester for the Hotels index enables autocomplete/suggestions
on the HotelName field only.
    // During indexing, HotelNames are indexed in patterns that support
autocomplete and suggested results.
    var suggestResult = await _searchClient.SuggestAsync<Hotel>(term, "sg",
options).ConfigureAwait(false);

    // Convert the suggest query results to a list that can be displayed in
the client.
    List<string> suggestions = suggestResult.Value.Results.Select(x =>
x.Text).ToList();

    // Return the list of suggestions.
    return new JsonResult(suggestions);
}
```

The SuggestAsync function takes two parameters that determine whether hit highlights are returned or fuzzy matching is used in addition to the search term input. Up to eight matches can be included in suggested results. The method creates a SuggestOptions object, which is then passed to the Suggest API. The result is then converted to JSON so it can be shown in the client.

Autocomplete

So far, the search UX code has been centered on suggestions. The next code block shows autocomplete, using the XDSoft jQuery UI Autocomplete function, passing in a

request for Azure Cognitive Search autocomplete. As with the suggestions, in a C# application, code that supports user interaction goes in **index.cshtml**.

JavaScript

```
$(function () {
    // using modified jQuery Autocomplete plugin v1.2.8
https://xdsoft.net/jqplugins/autocomplete/
    // $.autocomplete -> $.autocompleteInline
    $("#searchbox1").autocompleteInline({
        appendMethod: "replace",
        source: [
            function (text, add) {
                if (!text) {
                    return;
                }

                $.getJSON("/home/autocomplete?term=" + text, function (data)
{
                    if (data && data.length > 0) {
                        currentSuggestion2 = data[0];
                        add(data);
                    }
                });
            }
        ]
    });

    // complete on TAB and clear on ESC
    $("#searchbox1").keydown(function (evt) {
        if (evt.keyCode === 9 /* TAB */ && currentSuggestion2) {
            $("#searchbox1").val(currentSuggestion2);
            return false;
        } else if (evt.keyCode === 27 /* ESC */) {
            currentSuggestion2 = "";
            $("#searchbox1").val("");
        }
    });
});
```

Autocomplete function

Autocomplete is based on the AutocompleteAsync method. As with suggestions, this code block would go in the **HomeController.cs** file.

C#

```
public async Task<ActionResult> AutoCompleteAsync(string term)
{
    InitSearch();
```

```
    // Setup the autocomplete parameters.
    var ap = new AutocompleteOptions()
    {
        Mode = AutocompleteMode.OneTermWithContext,
        Size = 6
    };
    var autocompleteResult = await _searchClient.AutocompleteAsync(term,
"sg", ap).ConfigureAwait(false);

    // Convert the autocompleteResult results to a list that can be
displayed in the client.
    List<string> autocomplete = autocompleteResult.Value.Results.Select(x =>
x.Text).ToList();

    return new JsonResult(autocomplete);
}
```

The Autocomplete function takes the search term input. The method creates an AutoCompleteParameters object. The result is then converted to JSON so it can be shown in the client.

Next steps

The following tutorial demonstrates a search-as-you-type experience.

Add search to a web site (JavaScript)

Additional resources

Documentation

Create a suggester - Azure Cognitive Search

Enable type-ahead query actions in Azure Cognitive Search by creating suggesters and formulating requests that invoke autocomplete or autosuggested query terms.

Tutorial: create a custom analyzer - Azure Cognitive Search

Learn how to build a custom analyzer to improve the quality of search results in Azure Cognitive Search.

Autocomplete (Azure Cognitive Search REST API)

Send queries and obtain results based on partial inputs.

Map enrichments in indexers - Azure Cognitive Search

Export the enriched content created by a skillset by mapping its output fields to fields in a search index.

Add spell check to queries - Azure Cognitive Search

Attach spelling correction to the query pipeline, to fix typos on query terms before executing the query.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Show 5 more

Filters in Azure Cognitive Search

Article • 10/28/2022 • 8 minutes to read

A *filter* provides value-based criteria for including or excluding content before query execution. For example, including or excluding documents based on dates, locations, or language. Filters are specified on individual fields. A field definition must be attributed as "filterable" if you want to use it in filter expressions.

A filter is specified using OData filter expression syntax. In contrast with full text search, a filter succeeds only if the match is exact.

When to use a filter

Filters are foundational to several search experiences, including "find near me" geospatial search, faceted navigation, and security filters that show only those documents a user is allowed to see. If you implement any one of these experiences, a filter is required. It's the filter attached to the search query that provides the geolocation coordinates, the facet category selected by the user, or the security ID of the requestor.

Common scenarios include the following:

- Slice search results based on content in the index. Given a schema with hotel location, categories, and amenities, you might create a filter to explicitly match on criteria (in Seattle, on the water, with a view).
- Implement a search experience comes with a filter dependency:
  - Faceted navigation uses a filter to pass back the facet category selected by the user.
  - Geospatial search uses a filter to pass coordinates of the current location in "find near me" apps and functions that match within an area or by distance.
  - Security filters pass security identifiers as filter criteria, where a match in the index serves as a proxy for access rights to the document.
- Do a "numbers search". Numeric fields are retrievable and can appear in search results, but they aren't searchable (subject to full text search) individually. If you need selection criteria based on numeric data, use a filter.

How filters are executed

At query time, a filter parser accepts criteria as input, converts the expression into atomic Boolean expressions represented as a tree, and then evaluates the filter tree over

filterable fields in an index.

Filtering occurs in tandem with search, qualifying which documents to include in downstream processing for document retrieval and relevance scoring. When paired with a search string, the filter effectively reduces the recall set of the subsequent search operation. When used alone (for example, when the query string is empty where `search=*`), the filter criteria is the sole input.

Defining filters

Filters are OData expressions, articulated in the filter syntax supported by Cognitive Search.

You can specify one filter for each **search** operation, but the filter itself can include multiple fields, multiple criteria, and if you use an **ismatch** function, multiple full-text search expressions. In a multi-part filter expression, you can specify predicates in any order (subject to the rules of operator precedence). There's no appreciable gain in performance if you try to rearrange predicates in a particular sequence.

One of the limits on a filter expression is the maximum size limit of the request. The entire request, inclusive of the filter, can be a maximum of 16 MB for POST, or 8 KB for GET. There's also a limit on the number of clauses in your filter expression. A good rule of thumb is that if you have hundreds of clauses, you are at risk of running into the limit. We recommend designing your application in such a way that it doesn't generate filters of unbounded size.

The following examples represent prototypical filter definitions in several APIs.

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels/docs/search?
api-version=2020-06-30
{
    "search": "*",
    "filter": "Rooms/any(room: room/BaseRate lt 150.0)",
    "select": "HotelId, HotelName, Rooms/Description, Rooms/BaseRate"
}
```

C#

```
    parameters =
        new SearchParameters()
        {
            Filter = "Rooms/any(room: room/BaseRate lt 150.0)",
            Select = new[] { "HotelId", "HotelName", "Rooms/Description"
```

```
,"Rooms/BaseRate"}
        };

    var results = searchIndexClient.Documents.Search("*", parameters);
```

Filter patterns

The following examples illustrate several usage patterns for filter scenarios. For more ideas, see OData expression syntax > Examples.

- Standalone **$filter**, without a query string, useful when the filter expression is able to fully qualify documents of interest. Without a query string, there's no lexical or linguistic analysis, no scoring, and no ranking. Notice the search string is just an asterisk, which means "match all documents".

  JSON

  ```
  {
    "search": "*",
    "filter": "Rooms/any(room: room/BaseRate ge 60 and room/BaseRate lt
  300) and Address/City eq 'Honolulu"
  }
  ```

- Combination of query string and **$filter**, where the filter creates the subset, and the query string provides the term inputs for full text search over the filtered subset. The addition of terms (walking distance theaters) introduces search scores in the results, where documents that best match the terms are ranked higher. Using a filter with a query string is the most common usage pattern.

  JSON

  ```
  {
    "search": "walking distance theaters",
    "filter": "Rooms/any(room: room/BaseRate ge 60 and room/BaseRate lt
  300) and Address/City eq 'Seattle'"
  }
  ```

- Compound queries, separated by "or", each with its own filter criteria (for example, 'beagles' in 'dog' or 'siamese' in 'cat'). Expressions combined with `or` are evaluated individually, with the union of documents matching each expression sent back in the response. This usage pattern is achieved through the `search.ismatchscoring` function. You can also use the non-scoring version, `search.ismatch`.

```HTTP
# Match on hostels rated higher than 4 OR 5-star motels.
$filter=search.ismatchscoring('hostel') and Rating ge 4 or
search.ismatchscoring('motel') and Rating eq 5

# Match on 'luxury' or 'high-end' in the description field OR on
category exactly equal to 'Luxury'.
$filter=search.ismatchscoring('luxury | high-end', 'Description') or
Category eq 'Luxury'&$count=true
```

It's also possible to combine full-text search via `search.ismatchscoring` with filters using `and` instead of `or`, but this is functionally equivalent to using the `search` and `$filter` parameters in a search request. For example, the following two queries produce the same result:

```HTTP
$filter=search.ismatchscoring('pool') and Rating ge 4

search=pool&$filter=Rating ge 4
```

Field requirements for filtering

In the REST API, filterable is *on* by default for simple fields. Filterable fields increase index size; be sure to set `"filterable": false` for fields that you don't plan to actually use in a filter. For more information about settings for field definitions, see Create Index.

In the .NET SDK, the filterable is *off* by default. You can make a field filterable by setting the IsFilterable property of the corresponding SearchField object to `true`. In the example below, the attribute is set on the `BaseRate` property of a model class that maps to the index definition.

```C#
[IsFilterable, IsSortable, IsFacetable]
public double? BaseRate { get; set; }
```

Making an existing field filterable

You can't modify existing fields to make them filterable. Instead, you need to add a new field, or rebuild the index. For more information about rebuilding an index or repopulating fields, see How to rebuild an Azure Cognitive Search index.

Text filter fundamentals

Text filters match string fields against literal strings that you provide in the filter: `$filter=Category eq 'Resort and Spa'`

Unlike full-text search, there's no lexical analysis or word-breaking for text filters, so comparisons are for exact matches only. For example, assume a field *f* contains "sunny day", `$filter=f eq 'sunny'` doesn't match, but `$filter=f eq 'sunny day'` will.

Text strings are case-sensitive, which means text filters are case sensitive by default. For example, `$filter=f eq 'Sunny day'` won't find "sunny day". However, you can use a normalizer to make it so filtering isn't case sensitive.

Approaches for filtering on text

Approach	Description	When to use
search.in	A function that matches a field against a delimited list of strings.	Recommended for security filters and for any filters where many raw text values need to be matched with a string field. The **search.in** function is designed for speed and is much faster than explicitly comparing the field against each string using `eq` and `or`.
search.ismatch	A function that allows you to mix full-text search operations with strictly Boolean filter operations in the same filter expression.	Use **search.ismatch** (or its scoring equivalent, **search.ismatchscoring**) when you want multiple search-filter combinations in one request. You can also use it for a *contains* filter to filter on a partial string within a larger string.
$filter=field operator string	A user-defined expression composed of fields, operators, and values.	Use this when you want to find exact matches between a string field and a string value.

Numeric filter fundamentals

Numeric fields aren't `searchable` in the context of full text search. Only strings are subject to full text search. For example, if you enter 99.99 as a search term, you won't get back items priced at $99.99. Instead, you would see items that have the number 99 in string fields of the document. Thus, if you have numeric data, the assumption is that you'll use them for filters, including ranges, facets, groups, and so forth.

Documents that contain numeric fields (price, size, SKU, ID) provide those values in search results if the field is marked `retrievable`. The point here's that full text search itself isn't applicable to numeric field types.

Next steps

First, try **Search explorer** in the portal to submit queries with **$filter** parameters. The real-estate-sample index provides interesting results for the following filtered queries when you paste them into the search bar:

HTTP

```
# Geo-filter returning documents within 5 kilometers of Redmond, Washington
state
# Use $count=true to get a number of hits returned by the query
# Use $select to trim results, showing values for named fields only
# Use search=* for an empty query string. The filter is the sole input

search=*&$count=true&$select=description,city,postCode&$filter=geo.distance(
location,geography'POINT(-122.121513 47.673988)') le 5

# Numeric filters use comparison like greater than (gt), less than (lt), not
equal (ne)
# Include "and" to filter on multiple fields (baths and bed)
# Full text search is on John Leclerc, matching on John or Leclerc

search=John
Leclerc&$count=true&$select=source,city,postCode,baths,beds&$filter=baths gt
3 and beds gt 4

# Text filters can also use comparison operators
# Wrap text in single or double quotes and use the correct case
# Full text search is on John Leclerc, matching on John or Leclerc

search=John
Leclerc&$count=true&$select=source,city,postCode,baths,beds&$filter=city gt
'Seattle'
```

To work with more examples, see OData Filter Expression Syntax > Examples.

See also

- How full text search works in Azure Cognitive Search
- Search Documents REST API
- Simple query syntax
- Lucene query syntax

- Supported data types

Add faceted navigation to a search app

Article • 03/10/2023 • 11 minutes to read

Faceted navigation is used for self-directed drilldown filtering on query results in a search app, where your application offers form controls for scoping search to groups of documents (for example, categories or brands), and Azure Cognitive Search provides the data structures and filters to back the experience.

In this article, learn the basic steps for creating a faceted navigation structure in Azure Cognitive Search.

- ✓ Set field attributes in the index
- ✓ Structure the request and response
- ✓ Add navigation controls and filters in the presentation layer

Code in the presentation layer does the heavy lifting in a faceted navigation experience. The demos and samples listed at the end of this article provide working code that shows you how to bring everything together.

Faceted navigation in a search page

Facets are dynamic and returned on a query. A search response brings with it all of the facet categories used to navigate the documents in the result. The query executes first, and then facets are pulled from the current results and assembled into a faceted navigation structure.

In Cognitive Search, facets are one layer deep and can't be hierarchical. If you aren't familiar with faceted navigation structured, the following example shows one on the left. Counts indicate the number of matches for each facet. The same document can be represented in multiple facets.

Facets can help you find what you're looking for, while ensuring that you don't get zero results. As a developer, facets let you expose the most useful search criteria for navigating your search index.

Enable facets in the index

Faceting is enabled on a field-by-field basis in an index definition when you set the "facetable" attribute to true.

Although it's not strictly required, you should also set the "filterable" attribute so that you can build the necessary filters that back the faceted navigation experience in your search application.

The following example of the "hotels" sample index shows "facetable" and "filterable" on low cardinality fields that contain single values or short phrases: "Category", "Tags", "Rating".

JSON

```
{
  "name": "hotels",
  "fields": [
    { "name": "hotelId", "type": "Edm.String", "key": true, "searchable":
```

```
false, "sortable": false, "facetable": false },
    { "name": "Description", "type": "Edm.String", "filterable": false,
"sortable": false, "facetable": false },
    { "name": "HotelName", "type": "Edm.String", "facetable": false },
    { "name": "Category", "type": "Edm.String", "filterable": true,
"facetable": true },
    { "name": "Tags", "type": "Collection(Edm.String)", "filterable": true,
"facetable": true },
    { "name": "Rating", "type": "Edm.Int32", "filterable": true,
"facetable": true },
    { "name": "Location", "type": "Edm.GeographyPoint" }
  ]
}
```

Choosing fields

Facets can be calculated over single-value fields as well as collections. Fields that work best in faceted navigation have these characteristics:

- Low cardinality (a small number of distinct values that repeat throughout documents in your search corpus)
- Short descriptive values (one or two words) that will render nicely in a navigation tree

The contents of a field, and not the field itself, produces the facets in a faceted navigation structure. If the facet is a string field *Color*, facets will be blue, green, and any other value for that field.

As a best practice, check fields for null values, misspellings or case discrepancies, and single and plural versions of the same word. By default, filters and facets don't undergo lexical analysis or spell check, which means that all values of a "facetable" field are potential facets, even if the words differ by one character. Optionally, you can assign a normalizer to a "filterable" and "facetable" field to smooth out variations in casing and characters.

Defaults in REST and Azure SDKs

If you are using one of the Azure SDKs, your code must explicitly set the field attributes. In contrast, the REST API has defaults for field attributes based on the data type. The following data types are "filterable" and "facetable" by default:

- `Edm.String`
- `Edm.DateTimeOffset`
- `Edm.Boolean`

- `Edm.Int32`, `Edm.Int64`, `Edm.Double`
- Collections of any of the above types, for example `Collection(Edm.String)` or `Collection(Edm.Double)`

You can't use `Edm.GeographyPoint` or `Collection(Edm.GeographyPoint)` fields in faceted navigation. Facets work best on fields with low cardinality. Due to the resolution of geo-coordinates, it's rare that any two sets of coordinates will be equal in a given dataset. As such, facets aren't supported for geo-coordinates. You would need a city or region field to facet by location.

> **Tip**
>
> As a best practice for performance and storage optimization, turn faceting off for fields that should never be used as a facet. In particular, string fields for unique values, such as an ID or product name, should be set to `"facetable": false` to prevent their accidental (and ineffective) use in faceted navigation. This is especially true for the REST API that enables filters and facets by default.

Facet request and response

Facets are specified on the query, and the faceted navigation structure is returned at the top of the response. The structure of a request and response is fairly simple. In fact, the real work behind faceted navigation lies in the presentation layer, covered in a later section.

The following REST example is an unqualified query (`"search": "*"`) that is scoped to the entire index (see the built-in hotels sample). Facets are usually a list of fields, but this query shows just one for a more readable response below.

HTTP

```
POST https://{{service_name}}.search.windows.net/indexes/hotels/docs/search?
api-version={{api_version}}
{
    "search": "*",
    "queryType": "simple",
    "select": "",
    "searchFields": "",
    "filter": "",
    "facets": [ "Category"],
    "orderby": "",
    "count": true
}
```

It's useful to initialize a search page with an open query to completely fill in the faceted navigation structure. As soon as you pass query terms in the request, the faceted navigation structure will be scoped to just the matches in the results, rather than the entire index.

The response for the example above includes the faceted navigation structure at the top. The structure consists of "Category" values and a count of the hotels for each one. It's followed by the rest of the search results, trimmed here for brevity. This example works well for several reasons. The number of facets for this field fall under the limit (default is 10) so all of them appear, and every hotel in the index of 50 hotels is represented in exactly one of these categories.

JSON

```
{
    "@odata.context": "https://demo-search-
svc.search.windows.net/indexes('hotels')/$metadata#docs(*)",
    "@odata.count": 50,
    "@search.facets": {
        "Category": [
            {
                "count": 13,
                "value": "Budget"
            },
            {
                "count": 12,
                "value": "Resort and Spa"
            },
            {
                "count": 9,
                "value": "Luxury"
            },
            {
                "count": 7,
                "value": "Boutique"
            },
            {
                "count": 5,
                "value": "Suite"
            },
            {
                "count": 4,
                "value": "Extended-Stay"
            }
        ]
    },
    "value": [
        {
            "@search.score": 1.0,
            "HotelId": "1",
            "HotelName": "Secret Point Motel",
```

```
            "Description": "The hotel is ideally located on the main
commercial artery of the city in the heart of New York. A few minutes away
is Time's Square and the historic centre of the city, as well as other
places of interest that make New York one of America's most attractive and
cosmopolitan cities.",
            "Category": "Boutique",
            "Tags": [
                "pool",
                "air conditioning",
                "concierge"
            ],
            "ParkingIncluded": false,
  . . .
```

Facets syntax

A facet query parameter is set to a comma-delimited list of "facetable" fields and depending on the data type, can be further parameterized to set counts, sort orders, and ranges: `count:<integer>`, `sort:<>`, `interval:<integer>`, and `values:<list>`. For more detail about facet parameters, see "Query parameters" in the REST API.

HTTP

```
POST https://{{service_name}}.search.windows.net/indexes/hotels/docs/search?
api-version={{api_version}}
{
    "search": "*",
    "facets": [ "Category", "Tags,count:5", "Rating,values:1|2|3|4|5"],
    "count": true
}
```

For each faceted navigation tree, there's a default limit of the top ten facets. This default makes sense for navigation structures because it keeps the values list to a manageable size. You can override the default by assigning a value to "count". For example, `"Tags,count:5"` reduces the number of tags under the Tags section to the top five.

For Numeric and DateTime values only, you can explicitly set values on the facet field (for example, `facet=Rating,values:1|2|3|4|5`) to separate results into contiguous ranges (either ranges based on numeric values or time periods). Alternatively, you can add "interval", as in `facet=Rating,interval:1`.

Each range is built using 0 as a starting point, a value from the list as an endpoint, and then trimmed of the previous range to create discrete intervals.

Discrepancies in facet counts

Under certain circumstances, you might find that facet counts aren't fully accurate due to the sharding architecture. Every search index is spread across multiple shards, and each shard reports the top N facets by document count, which are then combined into a single result. Because it's just the top N facets for each shard, it's possible to miss or under-count matching documents in the facet response.

To guarantee accuracy, you can artificially inflate the count:<number> to a large number to force full reporting from each shard. You can specify `"count": "0"` for unlimited facets. Or, you can set "count" to a value that's greater than or equal to the number of unique values of the faceted field. For example, if you're faceting by a "size" field that has five unique values, you could set `"count:5"` to ensure all matches are represented in the facet response.

The tradeoff with this workaround is increased query latency, so use it only when necessary.

Presentation layer

In application code, the pattern is to use facet query parameters to return the faceted navigation structure along with facet results, plus a $filter expression. The filter expression handles the click event and further narrows the search result based on the facet selection.

Facet and filter combination

The following code snippet from the `JobsSearch.cs` file in the NYCJobs demo adds the selected Business Title to the filter if you select a value from the Business Title facet.

C#

```
if (businessTitleFacet != "")
  filter = "business_title eq '" + businessTitleFacet + "'";
```

Here's another example from the hotels sample. The following code snippet adds `categoyrFacet` to the filter if a user selects a value from the category facet.

C#

```
if (!String.IsNullOrEmpty(categoryFacet))
    filter = $"category eq '{categoryFacet}'";
```

HTML for faceted navigation

The following example, taken from the `index.cshtml` file of the NYCJobs sample application, shows the static HTML structure for displaying faceted navigation on the search results page. The list of facets is built or rebuilt dynamically when you submit a search term, or select or clear a facet.

HTML

```
<div class="widget sidebar-widget jobs-filter-widget">
  <h5 class="widget-title">Filter Results</h5>
    <p id="filterReset"></p>
    <div class="widget-content">

      <h6 id="businessTitleFacetTitle">Business Title</h6>
      <ul class="filter-list" id="business_title_facets">
      </ul>

      <h6>Location</h6>
      <ul class="filter-list" id="posting_type_facets">
      </ul>

      <h6>Posting Type</h6>
      <ul class="filter-list" id="posting_type_facets"></ul>

      <h6>Minimum Salary</h6>
      <ul class="filter-list" id="salary_range_facets">
      </ul>

  </div>
</div>
```

Build HTML dynamically

The following code snippet from the `index.cshtml` (also from NYCJobs demo) dynamically builds the HTML to display the first facet, Business Title. Similar functions dynamically build the HTML for the other facets. Each facet has a label and a count, which displays the number of items found for that facet result.

JavaScript

```
function UpdateBusinessTitleFacets(data) {
  var facetResultsHTML = '';
  for (var i = 0; i < data.length; i++) {
    facetResultsHTML += '<li><a href="javascript:void(0)"
onclick="ChooseBusinessTitleFacet(\'' + data[i].Value + '\');">' +
data[i].Value + ' (' + data[i].Count + ')</span></a></li>';
  }
```

```
  $("#business_title_facets").html(facetResultsHTML);
}
```

Tips for working with facets

This section is a collection of tips and workarounds that might be helpful.

Preserve a facet navigation structure asynchronously of filtered results

One of the challenges of faceted navigation in Azure Cognitive Search is that facets exist for current results only. In practice, it's common to retain a static set of facets so that the user can navigate in reverse, retracing steps to explore alternative paths through search content.

Although this is a common use case, it's not something the faceted navigation structure currently provides out-of-the-box. Developers who want static facets typically work around the limitation by issuing two filtered queries: one scoped to the results, the other used to create a static list of facets for navigation purposes.

Clear facets

When you design the search results page, remember to add a mechanism for clearing facets. If you add check boxes, you can easily see how to clear the filters. For other layouts, you might need a breadcrumb pattern or another creative approach. In the hotels C# sample, you can send an empty search to reset the page. In contrast, the NYCJobs sample application provides a clickable `[X]` after a selected facet to clear the facet, which is a stronger visual queue to the user.

Trim facet results with more filters

Facet results are documents found in the search results that match a facet term. In the following example, in search results for *cloud computing*, 254 items also have *internal specification* as a content type. Items aren't necessarily mutually exclusive. If an item meets the criteria of both filters, it's counted in each one. This duplication is possible when faceting on `Collection(Edm.String)` fields, which are often used to implement document tagging.

Output

```
Search term: "cloud computing"
Content type
   Internal specification (254)
   Video (10)
```

In general, if you find that facet results are consistently too large, we recommend that you add more filters to give users more options for narrowing the search.

A facet-only search experience

If your application uses faceted navigation exclusively (that is, no search box), you can mark the field as `searchable=false`, `filterable=true`, `facetable=true` to produce a more compact index. Your index won't include inverted indexes and there will be no text analysis or tokenization. Filters are made on exact matches at the character level.

Validate inputs at query-time

If you build the list of facets dynamically based on untrusted user input, validate that the names of the faceted fields are valid. Or, escape the names when building URLs by using either `Uri.EscapeDataString()` in .NET, or the equivalent in your platform of choice.

Demos and samples

Several samples include faceted navigation. This section has links to the samples and also notes which client library and language is used for each one.

Add search to web apps (React)

Tutorials and samples in C#, Python, and JavaScript include faceted navigation as well as filters, suggestions, and autocomplete. These samples use React for the presentation layer.

NYCJobs sample code and demo (Ajax)

The NYCJobs sample is an ASP.NET MVC application that uses Ajax in the presentation layer. It's available as a live demo app and as source code on Azure-Samples repo on GitHub.

Additional resources

🕮 Documentation

Text normalization for filters, facets, sort - Azure Cognitive Search

Specify normalizers to text fields in an index to customize the strict keyword matching behavior in filtering, faceting and sorting.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Indexer operations using Azure Cognitive Search REST APIs

Learn REST API calls used to create, delete, or update an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Add custom analyzers to string fields - Azure Cognitive Search

Configure text tokenizers and character filters to perform text analysis on strings during indexing and queries.

Show 5 more

Troubleshooting OData collection filters in Azure Cognitive Search

Article • 01/31/2023 • 11 minutes to read

To filter on collection fields in Azure Cognitive Search, you can use the any and all operators together with **lambda expressions**. A lambda expression is a sub-filter that is applied to each element of a collection.

Not every feature of filter expressions is available inside a lambda expression. Which features are available differs depending on the data type of the collection field that you want to filter. This can result in an error if you try to use a feature in a lambda expression that isn't supported in that context. If you're encountering such errors while trying to write a complex filter over collection fields, this article will help you troubleshoot the problem.

Common collection filter errors

The following table lists errors that you might encounter when trying to execute a collection filter. These errors happen when you use a feature of filter expressions that isn't supported inside a lambda expression. Each error gives some guidance on how you can rewrite your filter to avoid the error. The table also includes a link to the relevant section of this article that provides more information on how to avoid that error.

Error message	Situation	Details
The function 'ismatch' has no parameters bound to the range variable 's'. Only bound field references are supported inside lambda expressions ('any' or 'all'). However, you can change your filter so that the 'ismatch' function is outside the lambda expression and try again.	Using `search.ismatch` or `search.ismatchscoring` inside a lambda expression	Rules for filtering complex collections
Invalid lambda expression. Found a test for equality or inequality where the opposite was expected in a lambda expression that iterates over a field of type Collection(Edm.String). For 'any', use expressions of the form 'x eq y' or 'search.in(...)'. For 'all', use expressions of the form 'x ne y', 'not (x eq y)', or 'not search.in(...)'.	Filtering on a field of type `Collection(Edm.String)`	Rules for filtering string collections

Error message	Situation	Details
Invalid lambda expression. Found an unsupported form of complex Boolean expression. For 'any', use expressions that are 'ORs of ANDs', also known as Disjunctive Normal Form. For example: `(a and b) or (c and d)` where a, b, c, and d are comparison or equality sub-expressions. For 'all', use expressions that are 'ANDs of ORs', also known as Conjunctive Normal Form. For example: `(a or b) and (c or d)` where a, b, c, and d are comparison or inequality sub-expressions. Examples of comparison expressions: 'x gt 5', 'x le 2'. Example of an equality expression: 'x eq 5'. Example of an inequality expression: 'x ne 5'.	Filtering on fields of type `Collection(Edm.DateTimeOffset)`, `Collection(Edm.Double)`, `Collection(Edm.Int32)`, or `Collection(Edm.Int64)`	Rules for filtering comparable collections
Invalid lambda expression. Found an unsupported use of geo.distance() or geo.intersects() in a lambda expression that iterates over a field of type Collection(Edm.GeographyPoint). For 'any', make sure you compare geo.distance() using the 'lt' or 'le' operators and make sure that any usage of geo.intersects() isn't negated. For 'all', make sure you compare geo.distance() using the 'gt' or 'ge' operators and make sure that any usage of geo.intersects() is negated.	Filtering on a field of type `Collection(Edm.GeographyPoint)`	Rules for filtering GeographyPoint collections
Invalid lambda expression. Complex Boolean expressions aren't supported in lambda expressions that iterate over fields of type Collection(Edm.GeographyPoint). For 'any', join sub-expressions with 'or'; 'and' isn't supported. For 'all', oin sub-expressions with 'and'; 'or' isn't supported.	Filtering on fields of type `Collection(Edm.String)` or `Collection(Edm.GeographyPoint)`	Rules for filtering string collections Rules for filtering GeographyPoint collections
Invalid lambda expression. Found a comparison operator (one of 'lt', 'le', 'gt', or 'ge'). Only equality operators are allowed in lambda expressions that iterate over fields of type Collection(Edm.String). For 'any', se expressions of the form 'x eq y'. For 'all', use expressions of the form 'x ne y' or 'not (x eq y)'.	Filtering on a field of type `Collection(Edm.String)`	Rules for filtering string collections

How to write valid collection filters

The rules for writing valid collection filters are different for each data type. The following sections describe the rules by showing examples of which filter features are supported and which aren't:

- Rules for filtering string collections
- Rules for filtering Boolean collections
- Rules for filtering GeographyPoint collections
- Rules for filtering comparable collections
- Rules for filtering complex collections

Rules for filtering string collections

Inside lambda expressions for string collections, the only comparison operators that can be used are `eq` and `ne`.

> **Note**
>
> Azure Cognitive Search does not support the `lt`/`le`/`gt`/`ge` operators for strings, whether inside or outside a lambda expression.

The body of an `any` can only test for equality while the body of an `all` can only test for inequality.

It's also possible to combine multiple expressions via `or` in the body of an `any`, and via `and` in the body of an `all`. Since the `search.in` function is equivalent to combining equality checks with `or`, it's also allowed in the body of an `any`. Conversely, `not search.in` is allowed in the body of an `all`.

For example, these expressions are allowed:

- `tags/any(t: t eq 'books')`
- `tags/any(t: search.in(t, 'books, games, toys'))`
- `tags/all(t: t ne 'books')`
- `tags/all(t: not (t eq 'books'))`
- `tags/all(t: not search.in(t, 'books, games, toys'))`
- `tags/any(t: t eq 'books' or t eq 'games')`
- `tags/all(t: t ne 'books' and not (t eq 'games'))`

while these expressions aren't allowed:

- `tags/any(t: t ne 'books')`
- `tags/any(t: not search.in(t, 'books, games, toys'))`
- `tags/all(t: t eq 'books')`
- `tags/all(t: search.in(t, 'books, games, toys'))`
- `tags/any(t: t eq 'books' and t ne 'games')`
- `tags/all(t: t ne 'books' or not (t eq 'games'))`

Rules for filtering Boolean collections

The type `Edm.Boolean` supports only the `eq` and `ne` operators. As such, it doesn't make much sense to allow combining such clauses that check the same range variable with `and`/`or` since that would always lead to tautologies or contradictions.

Here are some examples of filters on Boolean collections that are allowed:

- `flags/any(f: f)`
- `flags/all(f: f)`
- `flags/any(f: f eq true)`
- `flags/any(f: f ne true)`
- `flags/all(f: not f)`
- `flags/all(f: not (f eq true))`

Unlike string collections, Boolean collections have no limits on which operator can be used in which type of lambda expression. Both `eq` and `ne` can be used in the body of `any` or `all`.

Expressions such as the following aren't allowed for Boolean collections:

- `flags/any(f: f or not f)`
- `flags/any(f: f or f)`
- `flags/all(f: f and not f)`
- `flags/all(f: f and f eq true)`

Rules for filtering GeographyPoint collections

Values of type `Edm.GeographyPoint` in a collection can't be compared directly to each other. Instead, they must be used as parameters to the `geo.distance` and `geo.intersects` functions. The `geo.distance` function in turn must be compared to a distance value using one of the comparison operators `lt`, `le`, `gt`, or `ge`. These rules also apply to non-collection Edm.GeographyPoint fields.

Like string collections, `Edm.GeographyPoint` collections have some rules for how the geo-spatial functions can be used and combined in the different types of lambda expressions:

- Which comparison operators you can use with the `geo.distance` function depends on the type of lambda expression. For `any`, you can use only `lt` or `le`. For `all`, you can use only `gt` or `ge`. You can negate expressions involving `geo.distance`, but you'll have to change the comparison operator (`geo.distance(...) lt x` becomes `not (geo.distance(...) ge x)` and `geo.distance(...) le x` becomes `not (geo.distance(...) gt x)`).
- In the body of an `all`, the `geo.intersects` function must be negated. Conversely, in the body of an `any`, the `geo.intersects` function must not be negated.
- In the body of an `any`, geo-spatial expressions can be combined using `or`. In the body of an `all`, such expressions can be combined using `and`.

The above limitations exist for similar reasons as the equality/inequality limitation on string collections. See Understanding OData collection filters in Azure Cognitive Search for a deeper look at these reasons.

Here are some examples of filters on `Edm.GeographyPoint` collections that are allowed:

- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') lt 10)`
- `locations/any(l: not (geo.distance(l, geography'POINT(-122 49)') ge 10) or geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') ge 10 and not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`

Expressions such as the following aren't allowed for `Edm.GeographyPoint` collections:

- `locations/any(l: l eq geography'POINT(-122 49)')`
- `locations/any(l: not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`
- `locations/all(l: geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`
- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') gt 10)`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') lt 10)`

- `locations/any(l: geo.distance(l, geography'POINT(-122 49)') lt 10 and geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') le 10 or not geo.intersects(l, geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'))`

Rules for filtering comparable collections

This section applies to all the following data types:

- `Collection(Edm.DateTimeOffset)`
- `Collection(Edm.Double)`
- `Collection(Edm.Int32)`
- `Collection(Edm.Int64)`

Types such as `Edm.Int32` and `Edm.DateTimeOffset` support all six of the comparison operators: `eq`, `ne`, `lt`, `le`, `gt`, and `ge`. Lambda expressions over collections of these types can contain simple expressions using any of these operators. This applies to both `any` and `all`. For example, these filters are allowed:

- `ratings/any(r: r ne 5)`
- `dates/any(d: d gt 2017-08-24T00:00:00Z)`
- `not margins/all(m: m eq 3.5)`

However, there are limitations on how such comparison expressions can be combined into more complex expressions inside a lambda expression:

- Rules for `any`:
  - Simple inequality expressions can't be usefully combined with any other expressions. For example, this expression is allowed:
    - `ratings/any(r: r ne 5)`

    but this expression isn't:
    - `ratings/any(r: r ne 5 and r gt 2)`

    and while this expression is allowed, it isn't useful because the conditions overlap:
    - `ratings/any(r: r ne 5 or r gt 7)`

- Simple comparison expressions involving `eq`, `lt`, `le`, `gt`, or `ge` can be combined with `and`/`or`. For example:
  - `ratings/any(r: r gt 2 and r le 5)`
  - `ratings/any(r: r le 5 or r gt 7)`
- Comparison expressions combined with `and` (conjunctions) can be further combined using `or`. This form is known in Boolean logic as "Disjunctive Normal Form" (DNF). For example:
  - `ratings/any(r: (r gt 2 and r le 5) or (r gt 7 and r lt 10))`

- Rules for `all`:
  - Simple equality expressions can't be usefully combined with any other expressions. For example, this expression is allowed:
    - `ratings/all(r: r eq 5)`

    but this expression isn't:
    - `ratings/all(r: r eq 5 or r le 2)`

    and while this expression is allowed, it isn't useful because the conditions overlap:
    - `ratings/all(r: r eq 5 and r le 7)`
  - Simple comparison expressions involving `ne`, `lt`, `le`, `gt`, or `ge` can be combined with `and`/`or`. For example:
    - `ratings/all(r: r gt 2 and r le 5)`
    - `ratings/all(r: r le 5 or r gt 7)`
  - Comparison expressions combined with `or` (disjunctions) can be further combined using `and`. This form is known in Boolean logic as "Conjunctive Normal Form" (CNF). For example:
    - `ratings/all(r: (r le 2 or gt 5) and (r lt 7 or r ge 10))`

Rules for filtering complex collections

Lambda expressions over complex collections support a much more flexible syntax than lambda expressions over collections of primitive types. You can use any filter construct inside such a lambda expression that you can use outside one, with only two exceptions.

First, the functions `search.ismatch` and `search.ismatchscoring` aren't supported inside lambda expressions. For more information, see Understanding OData collection filters in Azure Cognitive Search.

Second, referencing fields that aren't *bound* to the range variable (so-called *free variables*) isn't allowed. For example, consider the following two equivalent OData filter expressions:

1. `stores/any(s: s/amenities/any(a: a eq 'parking')) and details/margin gt 0.5`
2. `stores/any(s: s/amenities/any(a: a eq 'parking' and details/margin gt 0.5))`

The first expression will be allowed, while the second form will be rejected because `details/margin` isn't bound to the range variable `s`.

This rule also extends to expressions that have variables bound in an outer scope. Such variables are free with respect to the scope in which they appear. For example, the first expression is allowed, while the second equivalent expression isn't allowed because `s/name` is free with respect to the scope of the range variable `a`:

1. `stores/any(s: s/amenities/any(a: a eq 'parking') and s/name ne 'Flagship')`
2. `stores/any(s: s/amenities/any(a: a eq 'parking' and s/name ne 'Flagship'))`

This limitation shouldn't be a problem in practice since it's always possible to construct filters such that lambda expressions contain only bound variables.

Cheat sheet for collection filter rules

The following table summarizes the rules for constructing valid filters for each collection data type.

Data type	**Features allowed in lambda expressions with `any`**	**Features allowed in lambda expressions with `all`**
`Collection(Edm.ComplexType)`	Everything except `search.ismatch` and `search.ismatchscoring`	Same
`Collection(Edm.String)`	Comparisons with `eq` or `search.in` Combining sub-expressions with `or`	Comparisons with `ne` or `not search.in()` Combining sub-expressions with `and`
`Collection(Edm.Boolean)`	Comparisons with `eq` or `ne`	Same

Data type	Features allowed in lambda expressions with `any`	Features allowed in lambda expressions with `all`
`Collection(Edm.GeographyPoint)`	Using `geo.distance` with `lt` or `le` `geo.intersects` Combining sub-expressions with `or`	Using `geo.distance` with `gt` or `ge` `not geo.intersects(...)` Combining sub-expressions with `and`
`Collection(Edm.DateTimeOffset)`, `Collection(Edm.Double)`, `Collection(Edm.Int32)`, `Collection(Edm.Int64)`	Comparisons using `eq`, `ne`, `lt`, `gt`, `le`, or `ge` Combining comparisons with other sub-expressions using `or` Combining comparisons except `ne` with other sub-expressions using `and` Expressions using combinations of `and` and `or` in Disjunctive Normal Form (DNF)	Comparisons using `eq`, `ne`, `lt`, `gt`, `le`, or `ge` Combining comparisons with other sub-expressions using `and` Combining comparisons except `eq` with other sub-expressions using `or` Expressions using combinations of `and` and `or` in Conjunctive Normal Form (CNF)

For examples of how to construct valid filters for each case, see How to write valid collection filters.

If you write filters often, and understanding the rules from first principles would help you more than just memorizing them, see Understanding OData collection filters in Azure Cognitive Search.

Next steps

- Understanding OData collection filters in Azure Cognitive Search
- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Understanding OData collection filters in Azure Cognitive Search

Article • 01/31/2023 • 6 minutes to read

To filter on collection fields in Azure Cognitive Search, you can use the any and all operators together with **lambda expressions**. Lambda expressions are Boolean expressions that refer to a **range variable**. The `any` and `all` operators are analogous to a `for` loop in most programming languages, with the range variable taking the role of loop variable, and the lambda expression as the body of the loop. The range variable takes on the "current" value of the collection during iteration of the loop.

At least that's how it works conceptually. In reality, Azure Cognitive Search implements filters in a very different way to how `for` loops work. Ideally, this difference would be invisible to you, but in certain situations it isn't. The end result is that there are rules you have to follow when writing lambda expressions.

This article explains why the rules for collection filters exist by exploring how Azure Cognitive Search executes these filters. If you're writing advanced filters with complex lambda expressions, you may find this article helpful in building your understanding of what's possible in filters and why.

For information on what the rules for collection filters are, including examples, see Troubleshooting OData collection filters in Azure Cognitive Search.

Why collection filters are limited

There are three underlying reasons why not all filter features are supported for all types of collections:

1. Only certain operators are supported for certain data types. For example, it doesn't make sense to compare the Boolean values `true` and `false` using `lt`, `gt`, and so on.
2. Azure Cognitive Search doesn't support **correlated search** on fields of type `Collection(Edm.ComplexType)`.
3. Azure Cognitive Search uses inverted indexes to execute filters over all types of data, including collections.

The first reason is just a consequence of how the OData language and EDM type system are defined. The last two are explained in more detail in the rest of this article.

Correlated versus uncorrelated search

When applying multiple filter criteria over a collection of complex objects, the criteria are **correlated** since they apply to *each object in the collection*. For example, the following filter will return hotels that have at least one deluxe room with a rate less than 100:

odata-filter-expr

```
Rooms/any(room: room/Type eq 'Deluxe Room' and room/BaseRate lt 100)
```

If filtering was *uncorrelated*, the above filter might return hotels where one room is deluxe and a different room has a base rate less than 100. That wouldn't make sense, since both clauses of the lambda expression apply to the same range variable, namely `room`. This is why such filters are correlated.

However, for full-text search, there's no way to refer to a specific range variable. If you use fielded search to issue a full Lucene query like this one:

odata-filter-expr

```
Rooms/Type:deluxe AND Rooms/Description:"city view"
```

you may get hotels back where one room is deluxe, and a different room mentions "city view" in the description. For example, the document below with `Id` of `1` would match the query:

JSON

```
{
  "value": [
    {
      "Id": "1",
      "Rooms": [
        { "Type": "deluxe", "Description": "Large garden view suite" },
        { "Type": "standard", "Description": "Standard city view room" }
      ]
    },
    {
      "Id": "2",
      "Rooms": [
        { "Type": "deluxe", "Description": "Courtyard motel room" }
      ]
    }
  ]
}
```

The reason is that `Rooms/Type` refers to all the analyzed terms of the `Rooms/Type` field in the entire document, and similarly for `Rooms/Description`, as shown in the tables below.

How `Rooms/Type` is stored for full-text search:

Term in `Rooms/Type`	**Document IDs**
deluxe	1, 2
standard	1

How `Rooms/Description` is stored for full-text search:

Term in `Rooms/Description`	**Document IDs**
courtyard	2
city	1
garden	1
large	1
motel	2
room	1, 2
standard	1
suite	1
view	1

So unlike the filter above, which basically says "match documents where a room has `Type` equal to 'Deluxe Room' and **that same room** has `BaseRate` less than 100", the search query says "match documents where `Rooms/Type` has the term "deluxe" and `Rooms/Description` has the phrase "city view". There's no concept of individual rooms whose fields can be correlated in the latter case.

Inverted indexes and collections

You may have noticed that there are far fewer restrictions on lambda expressions over complex collections than there are for simple collections like `Collection(Edm.Int32)`, `Collection(Edm.GeographyPoint)`, and so on. This is because Azure Cognitive Search stores complex collections as actual collections of sub-documents, while simple collections aren't stored as collections at all.

For example, consider a filterable string collection field like `seasons` in an index for an online retailer. Some documents uploaded to this index might look like this:

JSON

```
{
  "value": [
    {
      "id": "1",
      "name": "Hiking boots",
      "seasons": ["spring", "summer", "fall"]
    },
    {
      "id": "2",
      "name": "Rain jacket",
      "seasons": ["spring", "fall", "winter"]
    },
    {
      "id": "3",
      "name": "Parka",
      "seasons": ["winter"]
    }
  ]
}
```

The values of the `seasons` field are stored in a structure called an **inverted index**, which looks something like this:

Term	Document IDs
spring	1, 2
summer	1
fall	1, 2
winter	2, 3

This data structure is designed to answer one question with great speed: In which documents does a given term appear? Answering this question works more like a plain equality check than a loop over a collection. In fact, this is why for string collections, Azure Cognitive Search only allows `eq` as a comparison operator inside a lambda expression for `any`.

Building up from equality, next we'll look at how it's possible to combine multiple equality checks on the same range variable with `or`. It works thanks to algebra and the distributive property of quantifiers. This expression:

odata-filter-expr

```
seasons/any(s: s eq 'winter' or s eq 'fall')
```

is equivalent to:

odata-filter-expr

```
seasons/any(s: s eq 'winter') or seasons/any(s: s eq 'fall')
```

and each of the two `any` sub-expressions can be efficiently executed using the inverted index. Also, thanks to the negation law of quantifiers, this expression:

odata-filter-expr

```
seasons/all(s: s ne 'winter' and s ne 'fall')
```

is equivalent to:

odata-filter-expr

```
not seasons/any(s: s eq 'winter' or s eq 'fall')
```

which is why it's possible to use `all` with `ne` and `and`.

> **Note**
>
> Although the details are beyond the scope of this document, these same principles extend to **distance and intersection tests for collections of geo-spatial points** as well. This is why, in `any`:
>
> - `geo.intersects` cannot be negated
> - `geo.distance` must be compared using `lt` or `le`
> - expressions must be combined with `or`, not `and`
>
> The converse rules apply for `all`.

A wider variety of expressions are allowed when filtering on collections of data types that support the `lt`, `gt`, `le`, and `ge` operators, such as `Collection(Edm.Int32)` for example. Specifically, you can use `and` as well as `or` in `any`, as long as the underlying comparison expressions are combined into **range comparisons** using `and`, which are then further combined using `or`. This structure of Boolean expressions is called

Disjunctive Normal Form (DNF), otherwise known as "ORs of ANDs". Conversely, lambda expressions for `all` for these data types must be in Conjunctive Normal Form (CNF), otherwise known as "ANDs of ORs". Azure Cognitive Search allows such range comparisons because it can execute them using inverted indexes efficiently, just like it can do fast term lookup for strings.

In summary, here are the rules of thumb for what's allowed in a lambda expression:

- Inside `any`, *positive checks* are always allowed, like equality, range comparisons, `geo.intersects`, or `geo.distance` compared with `lt` or `le` (think of "closeness" as being like equality when it comes to checking distance).
- Inside `any`, `or` is always allowed. You can use `and` only for data types that can express range checks, and only if you use ORs of ANDs (DNF).
- Inside `all`, the rules are reversed -- only *negative checks* are allowed, you can use `and` always, and you can use `or` only for range checks expressed as ANDs of ORs (CNF).

In practice, these are the types of filters you're most likely to use anyway. It's still helpful to understand the boundaries of what's possible though.

For specific examples of which kinds of filters are allowed and which aren't, see How to write valid collection filters.

Next steps

- Troubleshooting OData collection filters in Azure Cognitive Search
- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

Documentation

OData full-text search function reference - Azure Cognitive Search

OData full-text search functions, search.ismatch and search.ismatchscoring, in Azure Cognitive Search queries.

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search

queries.

OData search.score function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.score function in Azure Cognitive Search queries.

Troubleshooting OData collection filters - Azure Cognitive Search

Learn approaches for resolving OData collection filter errors in Azure Cognitive Search queries.

Data type map for indexers - Azure Cognitive Search

Source and destination field data type map combinations, used when crawling external data sources for data ingestion in an Azure Cognitive Search index.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

SearchClient.SearchAsync<T>(String, SearchOptions, CancellationToken) Method (Azure.Search.Documents) - Azure for .NET Developers

Searches for documents in the search index. Search Documents

Use simple Lucene query syntax - Azure Cognitive Search

Query examples demonstrating the simple syntax for full text search, filter search, and geo search against an Azure Cognitive Search index.

Show 5 more

Text normalization for case-insensitive filtering, faceting and sorting

Article • 10/21/2022 • 5 minutes to read

> **Important**
>
> This feature is in public preview under **Supplemental Terms of Use**. The **preview REST API** supports this feature.

In Azure Cognitive Search, a *normalizer* is a component that pre-processes text for keyword matching over fields marked as "filterable", "facetable", or "sortable". In contrast with full text "searchable" fields that are paired with text analyzers, content that's created for filter-facet-sort operations doesn't undergo analysis or tokenization. Omission of text analysis can produce unexpected results when casing and character differences show up.

By applying a normalizer, you can achieve light text transformations that improve results:

- Consistent casing (such as all lowercase or uppercase)
- Normalize accents and diacritics like ö or ê to ASCII equivalent characters "o" and "e"
- Map characters like `-` and whitespace into a user-specified character

Benefits of normalizers

Searching and retrieving documents from a search index requires matching the query input to the contents of the document. Matching is either over tokenized content, as is the case when you invoke "search", or over non-tokenized content if the request is a filter, facet, or orderby operation.

Because non-tokenized content is also not analyzed, small differences in the content are evaluated as distinctly different values. Consider the following examples:

- `$filter=City eq 'Las Vegas'` will only return documents that contain the exact text "Las Vegas" and exclude documents with "LAS VEGAS" and "las vegas", which is inadequate when the use-case requires all documents regardless of the casing.

- `search=*&facet=City,count:5` will return "Las Vegas", "LAS VEGAS" and "las vegas" as distinct values despite being the same city.

- `search=usa&$orderby=City` will return the cities in lexicographical order: "Las Vegas", "Seattle", "las vegas", even if the intent is to order the same cities together irrespective of the case.

A normalizer, which is invoked during indexing and query execution, adds light transformations that smooth out minor differences in text for filter, facet, and sort scenarios. In the previous examples, the variants of "Las Vegas" would be processed according to the normalizer you select (for example, all text is lower-cased) for more uniform results.

How to specify a normalizer

Normalizers are specified in an index definition, on a per-field basis, on text fields (`Edm.String` and `Collection(Edm.String)`) that have at least one of "filterable", "sortable", or "facetable" properties set to true. Setting a normalizer is optional and it's null by default. We recommend evaluating predefined normalizers before configuring a custom one.

Normalizers can only be specified when you add a new field to the index, so if possible, try to assess the normalization needs upfront and assign normalizers in the initial stages of development when dropping and recreating indexes is routine.

1. When creating a field definition in the index, set the "normalizer" property to one of the following values: a predefined normalizer such as "lowercase", or a custom normalizer (defined in the same index schema).

   JSON

   ```
   "fields": [
    {
      "name": "Description",
      "type": "Edm.String",
      "retrievable": true,
      "searchable": true,
      "filterable": true,
      "analyzer": "en.microsoft",
      "normalizer": "lowercase"
      ...
    }
   ]
   ```

2. Custom normalizers are defined in the "normalizers" section of the index first, and then assigned to the field definition as shown in the previous step. For more information, see Create Index and also Add custom normalizers.

```JSON
"fields": [
 {
   "name": "Description",
   "type": "Edm.String",
   "retrievable": true,
   "searchable": true,
   "analyzer": null,
   "normalizer": "my_custom_normalizer"
 },
```

> **Note**
>
> To change the normalizer of an existing field, you'll have to rebuild the index entirely (you cannot rebuild individual fields).

A good workaround for production indexes, where rebuilding indexes is costly, is to create a new field identical to the old one but with the new normalizer, and use it in place of the old one. Use Update Index to incorporate the new field and mergeOrUpload to populate it. Later, as part of planned index servicing, you can clean up the index to remove obsolete fields.

Predefined and custom normalizers

Azure Cognitive Search provides built-in normalizers for common use-cases along with the capability to customize as required.

Category	Description
Predefined normalizers	Provided out-of-the-box and can be used without any configuration.
Custom normalizers [1]	For advanced scenarios. Requires user-defined configuration of a combination of existing elements, consisting of char and token filters.

(1) Custom normalizers don't specify tokenizers since normalizers always produce a single token.

Normalizers reference

Predefined normalizers

Name	Description and Options
standard	Lowercases the text followed by asciifolding.
lowercase	Transforms characters to lowercase.
uppercase	Transforms characters to uppercase.
asciifolding	Transforms characters that aren't in the Basic Latin Unicode block to their ASCII equivalent, if one exists. For example, changing à to a.
elision	Removes elision from beginning of the tokens.

Supported char filters

Normalizers support two character filters that are identical to their counterparts in custom analyzer character filters:

- mapping
- pattern_replace

Supported token filters

The list below shows the token filters supported for normalizers and is a subset of the overall token filters used in custom analyzers.

- arabic_normalization
- asciifolding
- cjk_width
- elision
- german_normalization
- hindi_normalization
- indic_normalization
- persian_normalization
- scandinavian_normalization
- scandinavian_folding
- sorani_normalization
- lowercase
- uppercase

Add custom normalizers

Custom normalizers are defined within the index schema. The definition includes a name, a type, one or more character filters and token filters. The character filters and token filters are the building blocks for a custom normalizer and responsible for the processing of the text. These filters are applied from left to right.

The `token_filter_name_1` is the name of token filter, and `char_filter_name_1` and `char_filter_name_2` are the names of char filters (see supported token filters and supported char filterstables below for valid values).

JSON

```
"normalizers":(optional)[
   {
      "name":"name of normalizer",
      "@odata.type":"#Microsoft.Azure.Search.CustomNormalizer",
      "charFilters":[
         "char_filter_name_1",
         "char_filter_name_2"
      ],
      "tokenFilters":[
         "token_filter_name_1"
      ]
   }
],
"charFilters":(optional)[
   {
      "name":"char_filter_name_1",
      "@odata.type":"#char_filter_type",
      "option1": "value1",
      "option2": "value2",
      ...
   }
],
"tokenFilters":(optional)[
   {
      "name":"token_filter_name_1",
      "@odata.type":"#token_filter_type",
      "option1": "value1",
      "option2": "value2",
      ...
   }
]
```

Custom normalizers can be added during index creation or later by updating an existing one. Adding a custom normalizer to an existing index requires the "allowIndexDowntime" flag to be specified in Update Index and will cause the index to be unavailable for a few seconds.

Custom normalizer example

The example below illustrates a custom normalizer definition with corresponding character filters and token filters. Custom options for character filters and token filters are specified separately as named constructs, and then referenced in the normalizer definition as illustrated below.

- A custom normalizer named "my_custom_normalizer" is defined in the "normalizers" section of the index definition.
- The normalizer is composed of two character filters and three token filters: elision, lowercase, and customized asciifolding filter "my_asciifolding".
- The first character filter "map_dash" replaces all dashes with underscores while the second one "remove_whitespace" removes all spaces.

JSON

```
{
   "name":"myindex",
   "fields":[
      {
         "name":"id",
         "type":"Edm.String",
         "key":true,
         "searchable":false,
      },
      {
         "name":"city",
         "type":"Edm.String",
         "filterable": true,
         "facetable": true,
         "normalizer": "my_custom_normalizer"
      }
   ],
   "normalizers":[
      {
         "name":"my_custom_normalizer",
         "@odata.type":"#Microsoft.Azure.Search.CustomNormalizer",
         "charFilters":[
            "map_dash",
            "remove_whitespace"
         ],
         "tokenFilters":[
            "my_asciifolding",
            "elision",
            "lowercase",
         ]
      }
   ],
   "charFilters":[
```

```
        {
            "name":"map_dash",
            "@odata.type":"#Microsoft.Azure.Search.MappingCharFilter",
            "mappings":["-=>_"]
        },
        {
            "name":"remove_whitespace",
            "@odata.type":"#Microsoft.Azure.Search.MappingCharFilter",
            "mappings":["\\u0020=>"]
        }
    ],
    "tokenFilters":[
        {
            "name":"my_asciifolding",
            "@odata.type":"#Microsoft.Azure.Search.AsciiFoldingTokenFilter",
            "preserveOriginal":true
        }
    ]
}
```

See also

- Querying concepts in Azure Cognitive Search
- Analyzers for linguistic and text processing
- Search Documents REST API

Additional resources

🕮 Documentation

Azure Cognitive Search client library for .NET - Azure for .NET Developers

.NET samples - Azure Cognitive Search

Find Azure Cognitive Search demo C# code samples that use the .NET client libraries.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Model complex data types - Azure Cognitive Search

Nested or hierarchical data structures can be modeled in an Azure Cognitive Search index using ComplexType and Collections data types.

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Use full Lucene query syntax - Azure Cognitive Search

Query examples demonstrating the Lucene query syntax for fuzzy search, proximity search, term boosting, regular expression search, and wildcard searches in an Azure Cognitive Search index.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Add a faceted navigation category hierarchy - Azure Cognitive Search

Add faceted navigation for self-directed filtering in applications that integrate with Azure Cognitive Search.

Show 5 more

How to work with search results in Azure Cognitive Search

Article • 04/18/2023

This article explains how to work with a query response in Azure Cognitive Search. The structure of a response is determined by parameters in the query itself, as described in Search Documents (REST) or SearchResults Class (Azure for .NET).

Parameters on the query determine:

- Selection of fields within results
- Count of matches found in the index for the query
- Number of results in the response (up to 50, by default)
- Sort order
- Highlighting of terms within a result, matching on either the whole or partial term in the body

Result composition

Results are tabular, composed of fields of either all "retrievable" fields, or limited to just those fields specified in the `$select` parameters. Rows are the matching documents.

You can choose which fields are in search results. While a search document might have a large number of fields, typically only a few are needed to represent each document in results. On a query request, append `$select=<field list>` to specify which "retrievable" fields should appear in the response.

Pick fields that offer contrast and differentiation among documents, providing sufficient information to invite a click-through response on the part of the user. On an e-commerce site, it might be a product name, description, brand, color, size, price, and rating. For the built-in hotels-sample index, it might be the "select" fields in the following example:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
    {
      "search": "sandy beaches",
      "select": "HotelId, HotelName, Description, Rating, Address/City"
      "count": true
    }
```

> **Note**
>
> For images in results, such as a product photo or logo, store them outside of Azure Cognitive Search, but add a field in your index to reference the image URL in the search document. Sample indexes that demonstrate images in the results include the **realestate-sample-us** demo (a built-in sample dataset that you can build easily in the Import Data wizard), and the **New York City Jobs demo app**.

Tips for unexpected results

Occasionally, the substance and not the structure of results are unexpected. For example, you might find that some results appear to be duplicates, or a result that *should* appear near the top is positioned lower in the results. When query outcomes are unexpected, you can try these query modifications to see if results improve:

- Change `searchMode=any` (default) to `searchMode=all` to require matches on all criteria instead of any of the criteria. This is especially true when boolean operators are included the query.
- Experiment with different lexical analyzers or custom analyzers to see if it changes the query outcome. The default analyzer will break up hyphenated words and reduce words to root forms, which usually improves the robustness of a query response. However, if you need to preserve hyphens, or if strings include special characters, you might need to configure custom analyzers to ensure the index contains tokens in the right format. For more information, see Partial term search and patterns with special characters (hyphens, wildcard, regex, patterns).

Counting matches

The count parameter returns the number of documents in the index that are considered a match for the query. To return the count, add `$count=true` to the query request. There's no maximum value imposed by the search service. Depending on your query and the content of your documents, the count could be as high as every document in the index.

Count is accurate when the index is stable. If the system is actively adding, updating, or deleting documents, the count will be approximate, excluding any documents that aren't fully indexed.

Count won't be affected by routine maintenance or other workloads on the search service. However if you have multiple partitions and a single replica, you could experience short-term fluctuations in document count (several minutes) as the partitions are restarted.

> **Tip**
>
> To check indexing operations, you can confirm whether the index contains the expected number of documents by adding `$count=true` on an empty search `search=*` query. The result is the full count of documents in your index.
>
> When testing query syntax, `$count=true` can quickly tell you whether your modifications are returning greater or fewer results, which can be useful feedback.

Paging results

By default, the search engine returns up to the first 50 matches. The top 50 are determined by search score, assuming the query is full text search or semantic search. Otherwise, the top 50 are an arbitrary order for exact match queries (where uniform "@searchScore=1.0" indicates arbitrary ranking).

To control the paging of all documents returned in a result set, add `$top` and `$skip` parameters to the query request. The following list explains the logic.

- Return the first set of 15 matching documents plus a count of total matches: `GET /indexes/<INDEX-NAME>/docs?search=<QUERY STRING>&$top=15&$skip=0&$count=true`
- Return the second set, skipping the first 15 to get the next 15: `$top=15&$skip=15`. Repeat for the third set of 15: `$top=15&$skip=30`

The results of paginated queries aren't guaranteed to be stable if the underlying index is changing. Paging changes the value of `$skip` for each page, but each query is independent and operates on the current view of the data as it exists in the index at query time (in other words, there's no caching or snapshot of results, such as those found in a general purpose database).

Following is an example of how you might get duplicates. Assume an index with four documents:

```text
```

```
{ "id": "1", "rating": 5 }
{ "id": "2", "rating": 3 }
{ "id": "3", "rating": 2 }
{ "id": "4", "rating": 1 }
```

Now assume you want results returned two at a time, ordered by rating. You would execute this query to get the first page of results: `$top=2&$skip=0&$orderby=rating desc`, producing the following results:

```text
{ "id": "1", "rating": 5 }
{ "id": "2", "rating": 3 }
```

On the service, assume a fifth document is added to the index in between query calls: `{ "id": "5", "rating": 4 }`. Shortly thereafter, you execute a query to fetch the second page: `$top=2&$skip=2&$orderby=rating desc`, and get these results:

```text
{ "id": "2", "rating": 3 }
{ "id": "3", "rating": 2 }
```

Notice that document 2 is fetched twice. This is because the new document 5 has a greater value for rating, so it sorts before document 2 and lands on the first page. While this behavior might be unexpected, it's typical of how a search engine behaves.

Ordering results

In a full text search query, results can be ranked by:

- a search score
- a semantic reranker score
- a sort order on a "sortable" field

You can also boost any matches found in specific fields by adding a scoring profile.

Order by search score

For full text search queries, results are automatically ranked by a search score, calculated based on term frequency and proximity in a document (derived from TF-IDF), with higher scores going to documents having more or stronger matches on a search term.

The "@search.score" range is either unbounded, or 0 up to (but not including) 1.00 on older services.

For either algorithm, a "@search.score" equal to 1.00 indicates an unscored or unranked result set, where the 1.0 score is uniform across all results. Unscored results occur when the query form is fuzzy search, wildcard or regex queries, or an empty search (`search=*`). If you need to impose a ranking structure over unscored results, consider an `$orderby` expression to achieve that objective.

Order by the semantic reranker

If you're using semantic search, the "@search.rerankerScore" determines the sort order of your results.

The "@search.rerankerScore" range is 1 to 4.00, where a higher score indicates a stronger semantic match.

Order with $orderby

If consistent ordering is an application requirement, you can define an `$orderby` expression on a field. Only fields that are indexed as "sortable" can be used to order results.

Fields commonly used in an `$orderby` include rating, date, and location. Filtering by location requires that the filter expression calls the `geo.distance()` **function**, in addition to the field name.

Numeric fields (Edm.Double, Edm.Int32, Edm.Int64) are sorted in numeric order (for example, 1, 2, 10, 11, 20).

String fields (Edm.String, Edm.ComplexType subfields) are sorted in either ASCII sort order or Unicode sort order, depending on the language. You can't sort collections of any type.

- Numeric content in string fields is sorted alphabetically (1, 10, 11, 2, 20).
- Upper case strings are sorted ahead of lower case (APPLE, Apple, BANANA, Banana, apple, banana). You can assign a text normalizer to preprocess the text before sorting to change this behavior. Using the lowercase tokenizer on a field will have no effect on sorting behavior because Cognitive Search sorts on a non-analyzed copy of the field.
- Strings that lead with diacritics appear last (Äpfel, Öffnen, Üben)

Boost relevance using a scoring profile

Another approach that promotes order consistency is using a custom scoring profile. Scoring profiles give you more control over the ranking of items in search results, with the ability to boost matches found in specific fields. The extra scoring logic can help override minor differences among replicas because the search scores for each document are farther apart. We recommend the ranking algorithm for this approach.

Hit highlighting

Hit highlighting refers to text formatting (such as bold or yellow highlights) applied to matching terms in a result, making it easy to spot the match. Highlighting is useful for longer content fields, such as a description field, where the match isn't immediately obvious.

Notice that highlighting is applied to individual terms. There's no highlight capability for the contents of an entire field. If you want to highlight over a phrase, you'll have to provide the matching terms (or phrase) in a quote-enclosed query string. This technique is described further on in this section.

Hit highlighting instructions are provided on the query request. Queries that trigger query expansion in the engine, such as fuzzy and wildcard search, have limited support for hit highlighting.

Requirements for hit highlighting

- Fields must be Edm.String or Collection(Edm.String)
- Fields must be attributed at **searchable**

Specify highlighting in the request

To return highlighted terms, include the "highlight" parameter in the query request. The parameter is set to a comma-delimited list of fields.

By default, the format mark up is `<em>`, but you can override the tag using `highlightPreTag` and `highlightPostTag` parameters. Your client code handles the response (for example, applying a bold font or a yellow background).

HTTP

```
POST /indexes/good-books/docs/search?api-version=2020-06-30
    {
```

```
    "search": "divine secrets",
    "highlight": "title, original_title",
    "highlightPreTag": "<b>",
    "highlightPostTag": "</b>"
  }
```

By default, Azure Cognitive Search returns up to five highlights per field. You can adjust this number by appending a dash followed by an integer. For example, `"highlight": "description-10"` returns up to 10 highlighted terms on matching content in the "description" field.

Highlighted results

When highlighting is added to the query, the response includes an "@search.highlights" for each result so that your application code can target that structure. The list of fields specified for "highlight" are included in the response.

In a keyword search, each term is scanned for independently. A query for "divine secrets" will return matches on any document containing either term.

Keyword search highlighting

Within a highlighted field, formatting is applied to whole terms. For example, on a match against "The Divine Secrets of the Ya-Ya Sisterhood", formatting is applied to each term separately, even though they're consecutive.

JSON

```
"@odata.count": 39,
"value": [
    {
        "@search.score": 19.593246,
        "@search.highlights": {
            "original_title": [
                "<em>Divine</em> <em>Secrets</em> of the Ya-Ya Sisterhood"
            ],
            "title": [
                "<em>Divine</em> <em>Secrets</em> of the Ya-Ya Sisterhood"
            ]
        },
        "original_title": "Divine Secrets of the Ya-Ya Sisterhood",
        "title": "Divine Secrets of the Ya-Ya Sisterhood"
    },
    {
        "@search.score": 12.779835,
        "@search.highlights": {
            "original_title": [
                "<em>Divine</em> Madness"
            ],
            "title": [
                "<em>Divine</em> Madness (Cherub, #5)"
            ]
        },
        "original_title": "Divine Madness",
        "title": "Divine Madness (Cherub, #5)"
    },
    {
        "@search.score": 12.62534,
        "@search.highlights": {
            "original_title": [
                "Grave <em>Secrets</em>"
            ],
            "title": [
                "Grave <em>Secrets</em> (Temperance Brennan, #5)"
            ]
        },
        "original_title": "Grave Secrets",
        "title": "Grave Secrets (Temperance Brennan, #5)"
    }
```

Phrase search highlighting

Whole-term formatting applies even on a phrase search, where multiple terms are enclosed in double quotation marks. The following example is the same query, except that "divine search" is submitted as a quotation-enclosed phrase (some clients, such as Postman, require that you escape the interior quotation marks with a backslash `\"`):

HTTP

```
POST /indexes/good-books/docs/search?api-version=2020-06-30
    {
      "search": "\"divine secrets\"",,
      "select": "title,original_title",
      "highlight": "title",
      "highlightPreTag": "<b>",
      "highlightPostTag": "</b>",
      "count": true
    }
```

Because the criteria now specifies both terms, only one match is found in the search index. The response to the above query looks like this:

JSON

```
{
    "@odata.count": 1,
    "value": [
        {
            "@search.score": 19.593246,
            "@search.highlights": {
                "title": [
                    "<b>Divine</b> <b>Secrets</b> of the Ya-Ya Sisterhood"
                ]
            },
            "original_title": "Divine Secrets of the Ya-Ya Sisterhood",
            "title": "Divine Secrets of the Ya-Ya Sisterhood"
        }
    ]
}
```

Phrase highlighting on older services

Search services that were created before July 15, 2020 implement a different highlighting experience for phrase queries.

For the following examples, assume a query string that includes the quote-enclosed phrase "super bowl". Before July 2020, any term in the phrase is highlighted:

JSON

```
"@search.highlights": {
    "sentence": [
        "The <em>super</em> <em>bowl</em> is <em>super</em> awesome with a
<em>bowl</em> of chips"
    ]
```

For search services created after July 2020, only phrases that match the full phrase query will be returned in "@search.highlights":

JSON

```
"@search.highlights": {
    "sentence": [
        "The <em>super</em> <em>bowl</em> is super awesome with a bowl of
chips"
    ]
```

Next steps

To quickly generate a search page for your client, consider these options:

- Application Generator, in the portal, creates an HTML page with a search bar, faceted navigation, and results area that includes images.
- Add search to an ASP.NET Core (MVC) app is a tutorial and code sample that builds a functional client.
- Add search to web apps is a tutorial and code sample that uses the React JavaScript libraries for the user experience. The app is deployed using Azure Static Web Apps.

Return a semantic answer in Azure Cognitive Search

Article • 01/17/2023 • 8 minutes to read

> **Important**
>
> Semantic search is in public preview under **supplemental terms of use**. It's available through the Azure portal, preview REST API, and beta SDKs. This feature is billable (see **Availability and pricing**).

When invoking semantic ranking and captions, you can optionally extract content from the top-matching documents that "answers" the query directly. One or more answers can be included in the response, which you can then render on a search page to improve the user experience of your app.

In this article, learn how to request a semantic answer, unpack the response, and what content characteristics are most conducive to producing high-quality answers.

Prerequisites

All prerequisites that apply to semantic queries also apply to answers, including service tier and region.

- Query logic must include the semantic query parameters "queryType=semantic", plus the "answers" parameter. Required parameters are discussed in this article.
- Query strings entered by the user must be recognizable as a question (what, where, when, how).
- Search documents in the index must contain text having the characteristics of an answer, and that text must exist in one of the fields listed in the semantic configuration. For example, given a query "what is a hash table", if none of the fields in the semantic configuration contain passages that include "A hash table is ...", then it's unlikely an answer will be returned.

What is a semantic answer?

A semantic answer is a substructure of a semantic query response. It consists of one or more verbatim passages from a search document, formulated as an answer to a query

that looks like a question. To return an answer, phrases or sentences must exist in a search document that have the language characteristics of an answer, and the query itself must be posed as a question.

Cognitive Search uses a machine reading comprehension model to pick the best answer. The model produces a set of potential answers from the available content, and when it reaches a high enough confidence level, it will propose one as an answer.

Answers are returned as an independent, top-level object in the query response payload that you can choose to render on search pages, along side search results. Structurally, it's an array element within the response consisting of text, a document key, and a confidence score.

Formulate a REST query for "answers"

The approach for listing fields in priority order has changed, with "semanticConfiguration" replacing "searchFields". If you're currently using "searchFields", update your code to the 2021-04-30-Preview API version and use "semanticConfiguration" instead.

Semantic Configuration (recommended)

To return a semantic answer, the query must have the semantic "queryType", "queryLanguage", "semanticConfiguration", and the "answers" parameters. Specifying these parameters doesn't guarantee an answer, but the request must include them for answer processing to occur.

The "semanticConfiguration" parameter is required. It's defined in a search index, and then referenced in a query, as shown below.

JSON

```
{
    "search": "how do clouds form",
    "queryType": "semantic",
    "queryLanguage": "en-us",
    "semanticConfiguration": "my-semantic-config",
    "answers": "extractive|count-3",
    "captions": "extractive|highlight-true",
    "count": "true"
}
```

- A query string must not be null and should be formulated as question.

- "queryType" must be set to "semantic.
- "queryLanguage" must be one of the values from the supported languages list (REST API).
- A "semanticConfiguration" determines which string fields provide tokens to the extraction model. The same fields that produce captions also produce answers. See Create a semantic configuration for details.
- For "answers", parameter construction is `"answers": "extractive"`, where the default number of answers returned is one. You can increase the number of answers by adding a `count` as shown in the above example, up to a maximum of 10. Whether you need more than one answer depends on the user experience of your app, and how you want to render results.

Unpack an "answer" from the response

Answers are provided in the `"@search.answers"` array, which appears first in the query response. Each answer in the array will include:

- Document key
- Text or content of the answer, in plain text or with formatting
- Confidence score

If an answer is indeterminate, the response will show up as `"@search.answers": []`. The answers array is followed by the value array, which is the standard response in a semantic query.

Given the query "how do clouds form", the following example illustrates an answer:

JSON

```
{
    "@search.answers": [
        {
            "key": "4123",
            "text": "Sunlight heats the land all day, warming that moist air
and causing it to rise high into the   atmosphere until it cools and
condenses into water droplets. Clouds generally form where air is ascending
(over land in this case),   but not where it is descending (over the
river).",
            "highlights": "Sunlight heats the land all day, warming that
moist air and causing it to rise high into the   atmosphere until it cools
and condenses into water droplets. Clouds generally form<em> where air is
ascending</em> (over land in this case),   but not where it is<em>
```

```
descending</em> (over the river).",
            "score": 0.94639826
        }
    ],
    "value": [
        {
            "@search.score": 0.5479723,
            "@search.rerankerScore": 1.0321671911515296,
            "@search.captions": [
                {
                    "text": "Like all clouds, it forms when the air reaches
its dew point—the temperature at which an air mass is cool enough for its
water vapor to condense into liquid droplets. This false-color image shows
valley fog, which is common in the Pacific Northwest of North America.",
                    "highlights": "Like all<em> clouds</em>, it<em>
forms</em> when the air reaches its dew point—the temperature at    which an
air mass is cool enough for its water vapor to condense into liquid
droplets. This false-color image shows valley<em> fog</em>, which is common
in the Pacific Northwest of North America."
                }
            ],
            "title": "Earth Atmosphere",
            "content": "Fog is essentially a cloud lying on the ground. Like
all clouds, it forms when the air reaches its dew point—the temperature at
\n\nwhich an air mass is cool enough for its water vapor to condense into
liquid droplets.\n\nThis false-color image shows valley fog, which is common
in the Pacific Northwest of North America. On clear winter nights, the
\n\nground and overlying air cool off rapidly, especially at high
elevations. Cold air is denser than warm air, and it sinks down into the
\n\nvalleys. The moist air in the valleys gets chilled to its dew point, and
fog forms. If undisturbed by winds, such fog may persist for \n\ndays. The
Terra satellite captured this image of foggy valleys northeast of Vancouver
in February 2010.\n\n\n",
            "locations": [
                "Pacific Northwest",
                "North America",
                "Vancouver"
            ]
        }
    ]
}
```


When designing a search results page that includes answers, be sure to handle cases where answers are not found.

Within @search.answers:

- "**key**" is the document key or ID of the match. Given a document key, you can use Lookup Document API to retrieve any or all parts of the search document to include on the search page or a detail page.

- **"text"** and **"highlights"** provide identical content, in both plain text and with highlights.

  By default, highlights are styled as `<em>`, which you can override using the existing highlightPreTag and highlightPostTag parameters. As noted elsewhere, the substance of an answer is verbatim content from a search document. The extraction model looks for characteristics of an answer to find the appropriate content, but doesn't compose new language in the response.

- **"score"** is a confidence score that reflects the strength of the answer. If there are multiple answers in the response, this score is used to determine the order. Top answers and top captions can be derived from different search documents, where the top answer originates from one document, and the top caption from another, but in general you will see the same documents in the top positions within each array.

Answers are followed by the **"value"** array, which always includes scores, captions, and any fields that are retrievable by default. If you specified the select parameter, the "value" array is limited to the fields that you specified. See Configure semantic ranking for details.

Tips for producing high-quality answers

For best results, return semantic answers on a document corpus having the following characteristics:

- The "semanticConfiguration" must include fields that offer sufficient text in which an answer is likely to be found. Fields more likely to contain answers should be listed first in "prioritizedContentFields". Only verbatim text from a document can appear as an answer.

- Query strings must not be null (search=`*`) and the string should have the characteristics of a question, such as "what is" or "how to", as opposed to a keyword search consisting of terms or phrases in arbitrary order. If the query string doesn't appear to be a question, answer processing is skipped, even if the request specifies "answers" as a query parameter.

- Semantic extraction and summarization have limits over how many tokens per document can be analyzed in a timely fashion. In practical terms, if you have large documents that run into hundreds of pages, try to break up the content into smaller documents first.

Next steps

- Semantic search overview
- Semantic ranking algorithm
- Similarity ranking algorithm
- Configure semantic ranking

Additional resources

Documentation

Semantic ranking - Azure Cognitive Search

Learn how the semantic ranking algorithm works in Azure Cognitive Search.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Configure semantic search - Azure Cognitive Search

Set a semantic query type to attach the deep learning models of semantic search.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Relevance and scoring - Azure Cognitive Search

Explains the concepts of relevance and scoring in Azure Cognitive Search, and what a developer can do to customize the scoring result.

Add language analyzers to string fields - Azure Cognitive Search

Multi-lingual lexical analysis for non-English queries and indexes in Azure Cognitive Search.

Quickstart: Create a skillset in the Azure portal - Azure Cognitive Search

In this portal quickstart, use the Import Data wizard to add cognitive skills to an indexing pipeline to generate searchable text from images and unstructured documents. Skills in this quickstart include OCR, image analysis, and natural language processing.

Add scoring profiles - Azure Cognitive Search

Boost search relevance scores for Azure Cognitive Search results by adding scoring profiles to a search index.

Show 5 more

Use the "full" Lucene search syntax (advanced queries in Azure Cognitive Search)

Article • 01/20/2023 • 8 minutes to read

When constructing queries for Azure Cognitive Search, you can replace the default simple query parser with the more powerful Lucene query parser to formulate specialized and advanced query expressions.

The Lucene parser supports complex query formats, such as field-scoped queries, fuzzy search, infix and suffix wildcard search, proximity search, term boosting, and regular expression search. The additional power comes with additional processing requirements so you should expect a slightly longer execution time. In this article, you can step through examples demonstrating query operations based on full syntax.

> **Note**
>
> Many of the specialized query constructions enabled through the full Lucene query syntax are not **text-analyzed**, which can be surprising if you expect stemming or lemmatization. Lexical analysis is only performed on complete terms (a term query or phrase query). Query types with incomplete terms (prefix query, wildcard query, regex query, fuzzy query) are added directly to the query tree, bypassing the analysis stage. The only transformation performed on partial query terms is lowercasing.

Hotels sample index

The following queries are based on the hotels-sample-index, which you can create by following the instructions in this quickstart.

Example queries are articulated using the REST API and POST requests. You can paste and run them in Postman or another web client.

Request headers must have the following values:

Key	Value
Content-Type	application/json

Key	Value
api-key	`<your-search-service-api-key>`, either query or admin key

URI parameters must include your search service endpoint with the index name, docs collections, search command, and API version, similar to the following example:

HTTP

```
https://{{service-name}}.search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
```

Request body should be formed as valid JSON:

JSON

```
{
    "search": "*",
    "queryType": "full",
    "select": "HotelId, HotelName, Category, Tags, Description",
    "count": true
}
```

- "search" set to `*` is an unspecified query, equivalent to null or empty search. It's not especially useful, but it is the simplest search you can do, and it shows all retrievable fields in the index, with all values.
- "queryType" set to "full" invokes the full Lucene query parser and it's required for this syntax.
- "select" set to a comma-delimited list of fields is used for search result composition, including just those fields that are useful in the context of search results.
- "count" returns the number of documents matching the search criteria. On an empty search string, the count will be all documents in the index (50 in the case of hotels-sample-index).

Example 1: Fielded search

Fielded search scope individual, embedded search expressions to a specific field. This example searches for hotel names with the term "hotel" in them, but not "motel". You can specify multiple fields using AND.

When you use this query syntax, you can omit the "searchFields" parameter when the fields you want to query are in the search expression itself. If you include "searchFields" with fielded search, the `fieldName:searchExpression` always takes precedence over "searchFields".

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "HotelName:(hotel NOT motel) AND Category:'Resort and Spa'",
    "queryType": "full",
    "select": "HotelName, Category",
    "count": true
}
```

Response for this query should look similar to the following example, filtered on "Resort and Spa", returning hotels that include "hotel" in the name, while exlcuding results that include "motel" in the name.

JSON

```
"@odata.count": 4,
"value": [
    {
        "@search.score": 4.481559,
        "HotelName": "Nova Hotel & Spa",
        "Category": "Resort and Spa"
    },
    {
        "@search.score": 2.4524608,
        "HotelName": "King's Palace Hotel",
        "Category": "Resort and Spa"
    },
    {
        "@search.score": 2.3970203,
        "HotelName": "Triple Landscape Hotel",
        "Category": "Resort and Spa"
    },
    {
        "@search.score": 2.2953436,
        "HotelName": "Peaceful Market Hotel & Spa",
        "Category": "Resort and Spa"
    }
]
```

The search expression can be a single term or a phrase, or a more complex expression in parentheses, optionally with Boolean operators. Some examples include the following:

- `HotelName:(hotel NOT motel)`

- `Address/StateProvince:("WA" OR "CA")`
- `Tags:("free wifi" NOT "free parking") AND "coffee in lobby"`

Be sure to put a phrase within quotation marks if you want both strings to be evaluated as a single entity, as in this case searching for two distinct locations in the Address/StateProvince field. Depending on the client, you might need to escape (`\`) the quotation marks.

The field specified in `fieldName:searchExpression` must be a searchable field. See Create Index (REST API) for details on how field definitions are attributed.

Example 2: Fuzzy search

Fuzzy search matches on terms that are similar, including misspelled words. To do a fuzzy search, append the tilde `~` symbol at the end of a single word with an optional parameter, a value between 0 and 2, that specifies the edit distance. For example, `blue~` or `blue~1` would return blue, blues, and glue.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "Tags:conserge~",
    "queryType": "full",
    "select": "HotelName, Category, Tags",
    "searchFields": "HotelName, Category, Tags",
    "count": true
}
```

Response for this query resolves to "concierge" in the matching documents, trimmed for brevity:

JSON

```
"@odata.count": 12,
"value": [
    {
        "@search.score": 1.1832147,
        "HotelName": "Secret Point Motel",
        "Category": "Boutique",
        "Tags": [
            "pool",
            "air conditioning",
            "concierge"
        ]
    },
```

```
    {
        "@search.score": 1.1819803,
        "HotelName": "Twin Dome Motel",
        "Category": "Boutique",
        "Tags": [
            "pool",
            "free wifi",
            "concierge"
        ]
    },
    {
        "@search.score": 1.1773309,
        "HotelName": "Smile Hotel",
        "Category": "Suite",
        "Tags": [
            "view",
            "concierge",
            "laundry service"
        ]
    },
```

Phrases aren't supported directly but you can specify a fuzzy match on each term of a multi-part phrase, such as `search=Tags:landy~ AND sevic~`. This query expression finds 15 matches on "laundry service".

> ⓘ **Note**
>
> Fuzzy queries are not **analyzed**. Query types with incomplete terms (prefix query, wildcard query, regex query, fuzzy query) are added directly to the query tree, bypassing the analysis stage. The only transformation performed on partial query terms is lower casing.

Example 3: Proximity search

Proximity search finds terms that are near each other in a document. Insert a tilde "~" symbol at the end of a phrase followed by the number of words that create the proximity boundary.

This query searches for the terms "hotel" and "airport" within 5 words of each other in a document. The quotation marks are escaped (`\"`) to preserve the phrase:

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "Description: \"hotel airport\"~5",
```

```
    "queryType": "full",
    "select": "HotelName, Description",
    "searchFields": "HotelName, Description",
    "count": true
}
```

Response for this query should look similar to the following example:

JSON

```
"@odata.count": 2,
"value": [
    {
        "@search.score": 0.6331726,
        "HotelName": "Trails End Motel",
        "Description": "Only 8 miles from Downtown.  On-site bar/restaurant,
Free hot breakfast buffet, Free wireless internet, All non-smoking hotel.
Only 15 miles from airport."
    },
    {
        "@search.score": 0.43032226,
        "HotelName": "Catfish Creek Fishing Cabins",
        "Description": "Brand new mattresses and pillows.  Free airport
shuttle. Great hotel for your business needs. Comp WIFI, atrium lounge &
restaurant, 1 mile from light rail."
    }
]
```

Example 4: Term boosting

Term boosting refers to ranking a document higher if it contains the boosted term, relative to documents that do not contain the term. To boost a term, use the caret, `^`, symbol with a boost factor (a number) at the end of the term you are searching. The boost factor default is 1, and although it must be positive, it can be less than 1 (for example, 0.2). Term boosting differs from scoring profiles in that scoring profiles boost certain fields, rather than specific terms.

In this "before" query, search for "beach access" and notice that there are seven documents that match on one or both terms.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "beach access",
    "queryType": "full",
    "select": "HotelName, Description, Tags",
    "searchFields": "HotelName, Description, Tags",
```

```
    "count": true
}
```

In fact, there is only one document that matches on "access", and because it is the only match, it's placement is high (second position) even though the document is missing the term "beach".

JSON

```
"@odata.count": 7,
"value": [
    {
        "@search.score": 2.2723424,
        "HotelName": "Nova Hotel & Spa",
        "Description": "1 Mile from the airport.  Free WiFi, Outdoor Pool,
Complimentary Airport Shuttle, 6 miles from the beach & 10 miles from
downtown."
    },
    {
        "@search.score": 1.5507699,
        "HotelName": "Old Carrabelle Hotel",
        "Description": "Spacious rooms, glamorous suites and residences,
rooftop pool, walking access to shopping, dining, entertainment and the city
center."
    },
    {
        "@search.score": 1.5358944,
        "HotelName": "Whitefish Lodge & Suites",
        "Description": "Located on in the heart of the forest. Enjoy Warm
Weather, Beach Club Services, Natural Hot Springs, Airport Shuttle."
    },
    {
        "@search.score": 1.3433652,
        "HotelName": "Ocean Air Motel",
        "Description": "Oceanfront hotel overlooking the beach features
rooms with a private balcony and 2 indoor and outdoor pools. Various shops
and art entertainment are on the boardwalk, just steps away."
    },
```

In the "after" query, repeat the search, this time boosting results with the term "beach" over the term "access". A human readable version of the query is `search=Description:beach^2 access`. Depending on your client, you might need to express `^2` as `%5E2`.

After boosting the term "beach", the match on Old Carrabelle Hotel moves down to sixth place.

Example 5: Regex

A regular expression search finds a match based on the contents between forward slashes "/", as documented in the RegExp class.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "HotelName:/(Mo|Ho)tel/",
    "queryType": "full",
    "select": "HotelName",
    "count": true
}
```

Response for this query should look similar to the following example:

JSON

```
    "@odata.count": 22,
    "value": [
        {
            "@search.score": 1.0,
            "HotelName": "Days Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Triple Landscape Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Smile Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Pelham Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Sublime Cliff Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Twin Dome Motel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Nova Hotel & Spa"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Scarlet Harbor Hotel"
        },
```

> **Note**
>
> Regex queries are not **analyzed**. The only transformation performed on partial query terms is lower casing.

Example 6: Wildcard search

You can use generally recognized syntax for multiple (`*`) or single (`?`) character wildcard searches. Note the Lucene query parser supports the use of these symbols with a single term, and not a phrase.

In this query, search for hotel names that contain the prefix 'sc'. You cannot use a `*` or `?` symbol as the first character of a search.

HTTP

```
POST /indexes/hotel-samples-index/docs/search?api-version=2020-06-30
{
    "search": "HotelName:sc*",
    "queryType": "full",
    "select": "HotelName",
    "count": true
}
```

Response for this query should look similar to the following example:

JSON

```
    "@odata.count": 2,
    "value": [
        {
            "@search.score": 1.0,
            "HotelName": "Scarlet Harbor Hotel"
        },
        {
            "@search.score": 1.0,
            "HotelName": "Scottish Inn"
        }
    ]
```

> **Note**
>
> Wildcard queries are not **analyzed**. The only transformation performed on partial query terms is lower casing.

Next steps

Try specifying queries in code. The following links explain how to set up search queries using the Azure SDKs.

- Query your index using the .NET SDK
- Query your index using the Python SDK
- Query your index using the JavaScript SDK

Additional syntax reference, query architecture, and examples can be found in the following links:

- Lucene syntax query examples for building advanced queries
- How full text search works in Azure Cognitive Search
- Simple query syntax
- Full Lucene query syntax
- Filter syntax

Additional resources

🕮 Documentation

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Lookup Document (Azure Cognitive Search REST API)

Return a specific document by ID from an Azure Cognitive Search index.

Add, Update or Delete Documents (Azure Cognitive Search REST API)

Refresh content of an index by replacing, removing, or creating new documents.

Model complex data types - Azure Cognitive Search

Nested or hierarchical data structures can be modeled in an Azure Cognitive Search index using ComplexType and Collections data types.

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Data type map for indexers - Azure Cognitive Search

Source and destination field data type map combinations, used when crawling external data sources for data ingestion in an Azure Cognitive Search index.

Update Index (Azure Cognitive Search REST API)

Update an existing index definition in a search service.

Add custom analyzers to string fields - Azure Cognitive Search

Configure text tokenizers and character filters to perform text analysis on strings during indexing and queries.

Show 5 more

Partial term search and patterns with special characters (hyphens, wildcard, regex, patterns)

Article • 03/23/2023 • 10 minutes to read

A *partial term search* refers to queries consisting of term fragments, where instead of a whole term, you might have just the beginning, middle, or end of term (sometimes referred to as prefix, infix, or suffix queries). A partial term search might include a combination of fragments, often with special characters such as hyphens, dashes, or slashes that are part of the query string. Common use-cases include parts of a phone number, URL, codes, or hyphenated compound words.

Partial terms and special characters can be problematic if the index doesn't have a token that represents the text fragment you want to search for. During the lexical analysis phase of indexing (assuming the default standard analyzer), special characters are discarded, compound words are split up, and whitespace is deleted. If you're searching for a text fragment that was modified during lexical analysis, the query fails because no match is found. Consider this example: a phone number like `+1 (425) 703-6214` (tokenized as `"1"`, `"425"`, `"703"`, `"6214"`) won't show up in a `"3-62"` query because that content doesn't actually exist in the index.

The solution is to invoke an analyzer during indexing that preserves a complete string, including spaces and special characters if necessary, so that you can include the spaces and characters in your query string. Having a whole, un-tokenized string enables pattern matching for "starts with" or "ends with" queries, where the pattern you provide can be evaluated against a term that isn't transformed by lexical analysis.

If you need to support search scenarios that call for analyzed and non-analyzed content, consider creating two fields in your index, one for each scenario. One field undergoes lexical analysis. The second field stores an intact string, using a content-preserving analyzer that emits whole-string tokens for pattern matching.

> **Tip**
>
> If you are familiar with Postman and REST APIs, **download the query examples collection** to query partial terms and special characters described in this article.

About partial term search

Azure Cognitive Search scans for whole tokenized terms in the index and won't find a match on a partial term unless you include wildcard placeholder operators (`*` and `?`) , or format the query as a regular expression.

Partial terms are specified using these techniques:

- Regular expression queries can be any regular expression that is valid under Apache Lucene.
- Wildcard operators with prefix matching refers to a generally recognized pattern that includes the beginning of a term, followed by `*` or `?` suffix operators, such as `search=cap*` matching on "Cap'n Jack's Waterfront Inn" or "Gacc Capital". Prefixing matching is supported in both simple and full Lucene query syntax.
- Wildcard with infix and suffix matching places the `*` and `?` operators inside or at the beginning of a term, and requires regular expression syntax (where the expression is enclosed with forward slashes). For example, the query string (`search=/.*numeric.*/`) returns results on "alphanumeric" and "alphanumerical" as suffix and infix matches.

For regular expression, wildcard, and fuzzy search, analyzers aren't used at query time. For these query forms, which the parser detects by the presence of operators and delimiters, the query string is passed to the engine without lexical analysis. For these query forms, the analyzer specified on the field is ignored.

> **Note**
>
> When a partial query string includes characters, such as slashes in a URL fragment, you might need to add escape characters. In JSON, a forward slash `/` is escaped with a backward slash `\`. As such, `search=/.*microsoft.com\/azure\/.*/` is the syntax for the URL fragment "microsoft.com/azure/".

Solving partial/pattern search problems

When you need to search on fragments or patterns or special characters, you can override the default analyzer with a custom analyzer that operates under simpler tokenization rules, retaining the entire string in the index.

The approach looks like this:

1. Define a second field to store an intact version of the string (assuming you want analyzed and non-analyzed text at query time)

2. Evaluate and choose among the various analyzers that emit tokens at the right level of granularity
3. Assign the analyzer to the field
4. Build and test the index

> **Tip**
>
> Evaluating analyzers is an iterative process that requires frequent index rebuilds. You can make this step easier by using Postman, the REST APIs for **Create Index**, **Delete Index**,**Load Documents**, and **Search Documents**. For Load Documents, the request body should contain a small representative data set that you want to test (for example, a field with phone numbers or product codes). With these APIs in the same Postman collection, you can cycle through these steps quickly.

1 - Create a dedicated field

Analyzers determine how terms are tokenized in an index. Since analyzers are assigned on a per-field basis, you can create fields in your index to optimize for different scenarios. For example, you might define "featureCode" and "featureCodeRegex" to support regular full text search on the first, and advanced pattern matching on the second. The analyzers assigned to each field determine how the contents of each field are tokenized in the index.

JSON

```
{
  "name": "featureCode",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "analyzer": null
},
{
  "name": "featureCodeRegex",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "analyzer": "my_custom_analyzer"
},
```

2 - Set an analyzer

When choosing an analyzer that produces whole-term tokens, the following analyzers are common choices:

Analyzer	Behaviors
language analyzers	Preserves hyphens in compound words or strings, vowel mutations, and verb forms. If query patterns include dashes, using a language analyzer might be sufficient.
keyword	Content of the entire field is tokenized as a single term.
whitespace	Separates on white spaces only. Terms that include dashes or other characters are treated as a single token.
custom analyzer	(recommended) Creating a custom analyzer lets you specify both the tokenizer and token filter. The previous analyzers must be used as-is. A custom analyzer lets you pick which tokenizers and token filters to use. A recommended combination is the keyword tokenizer with a lower-case token filter. By itself, the built-in keyword analyzer doesn't lower-case any upper-case text, which can cause queries to fail. A custom analyzer gives you a mechanism for adding the lower-case token filter.

If you're using a web API test tool like Postman, you can add the Test Analyzer REST call to inspect tokenized output.

You must have a populated index to work with. Given an existing index and a field containing dashes or partial terms, you can try various analyzers over specific terms to see what tokens are emitted.

1. First, check the Standard analyzer to see how terms are tokenized by default.

   JSON

   ```
   {
   "text": "SVP10-NOR-00",
   "analyzer": "standard"
   }
   ```

2. Evaluate the response to see how the text is tokenized within the index. Notice how each term is lower-cased, hyphens removed, and substrings broken up into individual tokens. Only those queries that match on these tokens will return this document in the results. A query that includes "10-NOR" will fail.

   JSON

```
{
    "tokens": [
        {
            "token": "svp10",
            "startOffset": 0,
            "endOffset": 5,
            "position": 0
        },
        {
            "token": "nor",
            "startOffset": 6,
            "endOffset": 9,
            "position": 1
        },
        {
            "token": "00",
            "startOffset": 10,
            "endOffset": 12,
            "position": 2
        }
    ]
}
```

3. Now modify the request to use the `whitespace` or `keyword` analyzer:

JSON

```
{
"text": "SVP10-NOR-00",
"analyzer": "keyword"
}
```

4. This time, the response consists of a single token, upper-cased, with dashes preserved as a part of the string. If you need to search on a pattern or a partial term such as "10-NOR", the query engine now has the basis for finding a match.

JSON

```
{

    "tokens": [
        {
            "token": "SVP10-NOR-00",
            "startOffset": 0,
            "endOffset": 12,
            "position": 0
        }
    ]
}
```

> **Important**
>
> Be aware that query parsers often lower-case terms in a search expression when building the query tree. If you are using an analyzer that does not lower-case text inputs during indexing, and you are not getting expected results, this could be the reason. The solution is to add a lower-case token filter, as described in the "Use custom analyzers" section below.

3 - Configure an analyzer

Whether you're evaluating analyzers or moving forward with a specific configuration, you'll need to specify the analyzer on the field definition, and possibly configure the analyzer itself if you aren't using a built-in analyzer. When swapping analyzers, you typically need to rebuild the index (drop, recreate, and reload).

Use built-in analyzers

Built-in analyzers can be specified by name on an `analyzer` property of a field definition, with no extra configuration required in the index. The following example demonstrates how you would set the `whitespace` analyzer on a field.

For other scenarios and to learn more about other built-in analyzers, see Built-in analyzers.

JSON

```
{
  "name": "phoneNumber",
  "type": "Edm.String",
  "key": false,
  "retrievable": true,
  "searchable": true,
  "analyzer": "whitespace"
}
```

Use custom analyzers

If you're using a custom analyzer, define it in the index with a user-defined combination of tokenizer, token filter, with possible configuration settings. Next, reference it on a field definition, just as you would a built-in analyzer.

When the objective is whole-term tokenization, a custom analyzer that consists of a **keyword tokenizer** and **lower-case token filter** is recommended.

- The keyword tokenizer creates a single token for the entire contents of a field.
- The lowercase token filter transforms upper-case letters into lower-case text. Query parsers typically lowercase any uppercase text inputs. Lower-casing homogenizes the inputs with the tokenized terms.

The following example illustrates a custom analyzer that provides the keyword tokenizer and a lowercase token filter.

JSON

```
{
"fields": [
  {
  "name": "accountNumber",
  "analyzer":"myCustomAnalyzer",
  "type": "Edm.String",
  "searchable": true,
  "filterable": true,
  "retrievable": true,
  "sortable": false,
  "facetable": false
  }
],

"analyzers": [
  {
  "@odata.type":"#Microsoft.Azure.Search.CustomAnalyzer",
  "name":"myCustomAnalyzer",
  "charFilters":[],
  "tokenizer":"keyword_v2",
  "tokenFilters":["lowercase"]
  }
],
"tokenizers":[],
"charFilters": [],
"tokenFilters": []
}
```

Note

The `keyword_v2` tokenizer and `lowercase` token filter are known to the system and using their default configurations, which is why you can reference them by name without having to define them first.

4 - Build and test

Once you've defined an index with analyzers and field definitions that support your scenario, load documents that have representative strings so that you can test partial string queries.

The previous sections explained the logic. This section steps through each API you should call when testing your solution. As previously noted, if you use an interactive web test tool such as Postman, you can step through these tasks quickly.

- Delete Index removes an existing index of the same name so that you can recreate it.
- Create Index creates the index structure on your search service, including analyzer definitions and fields with an analyzer specification.
- Load Documents imports documents having the same structure as your index, as well as searchable content. After this step, your index is ready to query or test.
- Test Analyzer was introduced in Set an analyzer. Test some of the strings in your index using various analyzers to understand how terms are tokenized.
- Search Documents explains how to construct a query request, using either simple syntax or full Lucene syntax for wildcard and regular expressions.

  For partial term queries, such as querying "3-6214" to find a match on "+1 (425) 703-6214", you can use the simple syntax: `search=3-6214&queryType=simple`.

  For infix and suffix queries, such as querying "num" or "numeric to find a match on "alphanumeric", use the full Lucene syntax and a regular expression: `search=/.*num.*/&queryType=full`

Tune query performance

If you implement the recommended configuration that includes the keyword_v2 tokenizer and lower-case token filter, you might notice a decrease in query performance due to the extra token filter processing over existing tokens in your index.

The following example adds an EdgeNGramTokenFilter to make prefix matches faster. More tokens are generated for in 2-25 character combinations that include characters: (not only MS, MSF, MSFT, MSFT/, MSFT/S, MSFT/SQ, MSFT/SQL).

As you can imagine, the extra tokenization results in a larger index. If you have sufficient capacity to accommodate the larger index, this approach with its faster response time

might be a better solution.

JSON

```
{
"fields": [
  {
  "name": "accountNumber",
  "analyzer":"myCustomAnalyzer",
  "type": "Edm.String",
  "searchable": true,
  "filterable": true,
  "retrievable": true,
  "sortable": false,
  "facetable": false
  }
],

"analyzers": [
  {
  "@odata.type":"#Microsoft.Azure.Search.CustomAnalyzer",
  "name":"myCustomAnalyzer",
  "charFilters":[],
  "tokenizer":"keyword_v2",
  "tokenFilters":["lowercase", "my_edgeNGram"]
  }
],
"tokenizers":[],
"charFilters": [],
"tokenFilters": [
  {
  "@odata.type":"#Microsoft.Azure.Search.EdgeNGramTokenFilterV2",
  "name":"my_edgeNGram",
  "minGram": 2,
  "maxGram": 25,
  "side": "front"
  }
]
}
```

Next steps

This article explains how analyzers both contribute to query problems and solve query problems. As a next step, take a closer look at analyzer impact on indexing and query processing. In particular, consider using the Analyze Text API to return tokenized output so that you can see exactly what an analyzer is creating for your index.

- Tutorial: Create a custom analyzer for phone numbers
- Language analyzers

- Analyzers for text processing in Azure Cognitive Search
- Analyze Text API (REST)
- How full text search works (query architecture)

Fuzzy search to correct misspellings and typos

Article • 04/21/2023

Azure Cognitive Search supports fuzzy search, a type of query that compensates for typos and misspelled terms in the input string. It does this by scanning for terms having a similar composition. Expanding search to cover near-matches has the effect of auto-correcting a typo when the discrepancy is just a few misplaced characters.

What is fuzzy search?

It's a query expansion exercise that produces a match on terms having a similar composition. When a fuzzy search is specified, the search engine builds a graph (based on deterministic finite automaton theory) of similarly composed terms, for all whole terms in the query. For example, if your query includes three terms "university of washington", a graph is created for every term in the query `search=university~ of~ washington~` (there's no stop-word removal in fuzzy search, so "of" gets a graph).

The graph consists of up to 50 expansions, or permutations, of each term, capturing both correct and incorrect variants in the process. The engine then returns the topmost relevant matches in the response.

For a term like "university", the graph might have "unversty, universty, university, universe, inverse". Any documents that match on those in the graph are included in results. In contrast with other queries that analyze the text to handle different forms of the same word ("mice" and "mouse"), the comparisons in a fuzzy query are taken at face value without any linguistic analysis on the text. "Universe" and "inverse", which are semantically different, will match because the syntactic discrepancies are small.

A match succeeds if the discrepancies are limited to two or fewer edits, where an edit is an inserted, deleted, substituted, or transposed character. The string correction algorithm that specifies the differential is the Damerau-Levenshtein distance metric. It's described as the "minimum number of operations (insertions, deletions, substitutions, or transpositions of two adjacent characters) required to change one word into the other".

In Azure Cognitive Search:

- Fuzzy query applies to whole terms. Phrases aren't supported directly but you can specify a fuzzy match on each term of a multi-part phrase through AND

constructions. For example, `search=dr~ AND cleanin~`. This query expression finds matches on "dry cleaning".

- The default distance of an edit is 2. A value of `~0` signifies no expansion (only the exact term is considered a match), but you could specify `~1` for one degree of difference, or one edit.
- A fuzzy query can expand a term up to 50 permutations. This limit isn't configurable, but you can effectively reduce the number of expansions by decreasing the edit distance to 1.
- Responses consist of documents containing a relevant match (up to 50).

During query processing, fuzzy queries don't undergo lexical analysis. The query input is added directly to the query tree and expanded to create a graph of terms. The only transformation performed is lower casing.

Collectively, the graphs are submitted as match criteria against tokens in the index. As you can imagine, fuzzy search is inherently slower than other query forms. The size and complexity of your index can determine whether the benefits are enough to offset the latency of the response.

> ⓘ **Note**
>
> Because fuzzy search tends to be slow, it might be worthwhile to investigate alternatives such as n-gram indexing, with its progression of short character sequences (two and three character sequences for bigram and trigram tokens). Depending on your language and query surface, n-gram might give you better performance. The trade off is that n-gram indexing is very storage intensive and generates much bigger indexes.
>
> Another alternative, which you could consider if you want to handle just the most egregious cases, would be a **synonym map**. For example, mapping "search" to "serach, serch, sarch", or "retrieve" to "retreive".

Indexing for fuzzy search

String fields that are attributed as "searchable" are candidates for fuzzy search.

Analyzers aren't used to create an expansion graph, but that doesn't mean analyzers should be ignored in fuzzy search scenarios. Analyzers are important for tokenization

during indexing, where tokens in the inverted indexes are used for matching against the graph.

As always, if test queries aren't producing the matches you expect, experiment with different indexing analyzers. For example, try a language analyzer to see if you get better results. Some languages, particularly those with vowel mutations, can benefit from the inflection and irregular word forms generated by the Microsoft natural language processors. In some cases, using the right language analyzer can make a difference in whether a term is tokenized in a way that is compatible with the value provided by the user.

How to invoke fuzzy search

Fuzzy queries are constructed using the full Lucene query syntax, invoking the full Lucene query parser, and appending a tilde character `~` after each whole term entered by the user.

Here's an example of a query request that invokes fuzzy search. It includes four terms, two of which are misspelled:

HTTP

```
POST https://[service name].search.windows.net/indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "seatle~ waterfront~ view~ hotle~",
    "queryType": "full",
    "searchMode": "any",
    "searchFields": "HotelName, Description",
    "select": "HotelName, Description, Address/City,",
    "count": "true"
}
```

1. Set the query type to the full Lucene syntax (`queryType=full`).

2. Provide the query string where each term is followed by a tilde (`~`) operator at the end of each whole term (`search=<string>~`). An expansion graph will be created for every term in the query input.

   Include an optional parameter, a number between 0 and 2 (default), if you want to specify the edit distance (`~1`). For example, "blue~" or "blue~1" would return "blue", "blues", and "glue".

Optionally, you can improve query performance by scoping the request to specific fields. Use the `searchFields` parameter to specify which fields to search. You can also use the `select` property to specify which fields are returned in the query response.

Testing fuzzy search

For simple testing, we recommend Search explorer or Postman for iterating over a query expression. Both tools are interactive, which means you can quickly step through multiple variants of a term and evaluate the responses that come back.

When results are ambiguous, hit highlighting can help you identify the match in the response.

> **Note**
>
> The use of hit highlighting to identify fuzzy matches has limitations and only works for basic fuzzy search. If your index has scoring profiles, or if you layer the query with more syntax, hit highlighting might fail to identify the match.

Example 1: fuzzy search with the exact term

Assume the following string exists in a `"Description"` field in a search document: `"Test queries with special characters, plus strings for MSFT, SQL and Java."`

Start with a fuzzy search on "special" and add hit highlighting to the Description field:

Console

```
search=special~&highlight=Description
```

In the response, because you added hit highlighting, formatting is applied to "special" as the matching term.

Output

```
"@search.highlights": {
    "Description": [
        "Test queries with <em>special</em> characters, plus strings for
MSFT, SQL and Java."
    ]
```

Try the request again, misspelling "special" by taking out several letters ("pe"):

Console

```
search=scial~&highlight=Description
```

So far, no change to the response. Using the default of 2 degrees distance, removing two characters "pe" from "special" still allows for a successful match on that term.

Output

```
"@search.highlights": {
    "Description": [
        "Test queries with <em>special</em> characters, plus strings for
MSFT, SQL and Java."
    ]
```

Trying one more request, further modify the search term by taking out one last character for a total of three deletions (from "special" to "scal"):

Console

```
search=scal~&highlight=Description
```

Notice that the same response is returned, but now instead of matching on "special", the fuzzy match is on "SQL".

Output

```
        "@search.score": 0.4232868,
        "@search.highlights": {
            "Description": [
                "Mix of special characters, plus strings for MSFT,
<em>SQL</em>, 2019, Linux, Java."
            ]
```

The point of this expanded example is to illustrate the clarity that hit highlighting can bring to ambiguous results. In all cases, the same document is returned. Had you relied on document IDs to verify a match, you might have missed the shift from "special" to "SQL".

See also

- How full text search works in Azure Cognitive Search (query parsing architecture)
- Search explorer
- How to query in .NET

- How to query in REST

Configure relevance scoring

Article • 04/18/2023

In this article, you'll learn how to configure the similarity scoring algorithm used by Azure Cognitive Search. The BM25 scoring model has defaults for weighting term frequency and document length. You can customize these properties if the defaults aren't suited to your content.

Configuration changes are scoped to individual indexes, which means you can adjust relevance scoring based on the characteristics of each index.

Default scoring algorithm

Depending on the age of your search service, Azure Cognitive Search supports two similarity scoring algorithms for a full text search query:

- Okapi BM25 algorithm (after July 15, 2020)
- Classic similarity algorithm (before July 15, 2020)

BM25 ranking is the default because it tends to produce search rankings that align better with user expectations. It includes parameters for tuning results based on factors such as document size. For search services created after July 2020, BM25 is the only scoring algorithm. If you try to set "similarity" to ClassicSimilarity on a new service, an HTTP 400 error will be returned because that algorithm is not supported by the service.

For older services, classic similarity remains the default algorithm. Older services can upgrade to BM25 on a per-index basis. When switching from classic to BM25, you can expect to see some differences how search results are ordered.

Set BM25 parameters

BM25 similarity adds two parameters to control the relevance score calculation.

1. Formulate a Create or Update Index request as illustrated by the following example.

```
HTTP

PUT [service-name].search.windows.net/indexes/[index-name]?api-
version=2020-06-30&allowIndexDowntime=true
{
    "similarity": {
```

```
        "@odata.type": "#Microsoft.Azure.Search.BM25Similarity",
        "b" : 0.75,
        "k1" : 1.2
    }
}
```

2. Set "b" and "k1" to custom values. See the property descriptions in the next section for details.

3. If the index is live, append the "allowIndexDowntime=true" URI parameter on the request.

   Because Cognitive Search won't allow updates to a live index, you'll need to take the index offline so that the parameters can be added. Indexing and query requests will fail while the index is offline. The duration of the outage is the amount of time it takes to update the index, usually no more than several seconds. When the update is complete, the index comes back automatically.

4. Send the request.

BM25 property descriptions

Property	Type	Description
k1	number	Controls the scaling function between the term frequency of each matching terms to the final relevance score of a document-query pair. Values are usually 0.0 to 3.0, with 1.2 as the default. A value of 0.0 represents a "binary model", where the contribution of a single matching term is the same for all matching documents, regardless of how many times that term appears in the text, while a larger k1 value allows the score to continue to increase as more instances of the same term is found in the document. Using a higher k1 value can be important in cases where we expect multiple terms to be part of a search query. In those cases, we might want to favor documents that match many of the different query terms being searched over documents that only match a single one, multiple times. For example, when querying the index for documents containing the terms "Apollo Spaceflight", we might want to lower the score of an article about Greek Mythology that contains the term "Apollo" a few dozen times, without mentions of "Spaceflight", compared to another article that explicitly mentions both "Apollo" and "Spaceflight" a handful of times only.

Property	Type	Description
b	number	Controls how the length of a document affects the relevance score. Values are between 0 and 1, with 0.75 as the default. A value of 0.0 means the length of the document will not influence the score, while a value of 1.0 means the impact of term frequency on relevance score will be normalized by the document's length. Normalizing the term frequency by the document's length is useful in cases where we want to penalize longer documents. In some cases, longer documents (such as a complete novel), are more likely to contain many irrelevant terms, compared to much shorter documents.

Enable BM25 scoring on older services

If you're running a search service that was created from March 2014 through July 15, 2020, you can enable BM25 by setting a "similarity" property on new indexes. The property is only exposed on new indexes, so if want BM25 on an existing index, you must drop and rebuild the index with a "similarity" property set to "Microsoft.Azure.Search.BM25Similarity".

Once an index exists with a "similarity" property, you can switch between `BM25Similarity` or `ClassicSimilarity`.

The following links describe the Similarity property in the Azure SDKs.

Client library	Similarity property
.NET	SearchIndex.Similarity
Java	SearchIndex.setSimilarity
JavaScript	SearchIndex.Similarity
Python	similarity property on SearchIndex

REST example

You can also use the REST API. The following example creates a new index with the "similarity" property set to BM25:

```
HTTP
```

```
PUT [service-name].search.windows.net/indexes/[index name]?api-version=2020-06-30
{
    "name": "indexName",
    "fields": [
        {
            "name": "id",
            "type": "Edm.String",
            "key": true
        },
        {
            "name": "name",
            "type": "Edm.String",
            "searchable": true,
            "analyzer": "en.lucene"
        },
        ...
    ],
    "similarity": {
        "@odata.type": "#Microsoft.Azure.Search.BM25Similarity"
    }
}
```

See also

- Similarity and scoring in Azure Cognitive Search
- REST API Reference
- Add scoring profiles to your index
- Create Index API
- Azure Cognitive Search .NET SDK

Add scoring profiles to boost search scores

Article • 03/15/2023 • 15 minutes to read

In this article, you'll learn how to define a scoring profile for boosting search scores based on criteria.

Criteria can be a weighted field, such as when a match found in a "tags" field is more relevant than a match found in "descriptions". Criteria can also be a function, such as the `distance` function that favors results that are within a specified distance of the current location.

Scoring profiles are defined in a search index and invoked on query requests. You can create multiple profiles and then modify query logic to choose which one is used.

> **Note**
>
> Unfamiliar with relevance concepts? The following video segment fast-forwards to how scoring profiles work in Azure Cognitive Search. You can also visit **Relevance and scoring in Azure Cognitive Search** for more background.
> **https://www.youtube-nocookie.com/embed/Y_X6USgvB1g?version=3&start=463&end=970**

Scoring profile definition

A scoring profile is part of the index definition and is composed of weighted fields, functions, and parameters.

The following definition shows a simple profile named 'geo'. This example boosts results that have the search term in the hotelName field. It also uses the `distance` function to favor results that are within 10 kilometers of the current location. If someone searches on the term 'inn', and 'inn' happens to be part of the hotel name, documents that include hotels with 'inn' within a 10 KM radius of the current location will appear higher in the search results.

JSON

```
"scoringProfiles": [
  {
    "name":"geo",
```

```
    "text": {
      "weights": {
        "hotelName": 5
      }
    },
    "functions": [
      {
        "type": "distance",
        "boost": 5,
        "fieldName": "location",
        "interpolation": "logarithmic",
        "distance": {
          "referencePointParameter": "currentLocation",
          "boostingDistance": 10
        }
      }
    ]
  }
]
```

To use this scoring profile, your query is formulated to specify scoringProfile parameter in the request. If you're using the REST API, queries are specified through GET and POST requests.

HTTP

```
GET /indexes/hotels/docs?
search+inn&scoringProfile=geo&scoringParameter=currentLocation-
-122.123,44.77233&api-version=2020-06-30
```

Notice the syntax differences when using POST. In POST, "scoringParameters" is plural and it's an array.

HTTP

```
POST /indexes/hotels/docs&api-version=2020-06-30
{
    "search": "inn",
    "scoringProfile": "geo",
    "scoringParameters": [currentLocation--122.123,44.77233]
}
```

This query searches on the term "inn" and passes in the current location. Notice that this query includes other parameters, such as scoringParameter. Query parameters, including "scoringParameter", are described in Search Documents (REST API).

See the Extended example to review a more detailed example of a scoring profile.

How scores are computed

Scores are computed for full text search queries for ranking the most relevant matches and returning them at the top of the response. The overall score for each document is an aggregation of the individual scores for each field, where the individual score of each field is computed based on the term frequency and document frequency of the searched terms within that field (known as TF-IDF or term frequency-inverse document frequency).

You can use the featuresMode (preview) parameter to request extra scoring details with the search results (including the field level scores).

When to add scoring logic

You should create one or more scoring profiles when the default ranking behavior doesn't go far enough in meeting your business objectives. For example, you might decide that search relevance should favor newly added items. Likewise, you might have a field that contains profit margin, or some other field indicating revenue potential. Boosting results that are more meaningful to your users or the business is often the deciding factor in adoption of scoring profiles.

Relevancy-based ordering in a search page is also implemented through scoring profiles. Consider search results pages you've used in the past that let you sort by price, date, rating, or relevance. In Azure Cognitive Search, scoring profiles can be used to drive the 'relevance' option. The definition of relevance is user-defined, predicated on business objectives and the type of search experience you want to deliver.

Steps for adding a scoring profile

To implement custom scoring behavior, add a scoring profile to the schema that defines the index. You can have up to 100 scoring profiles within an index (see Service Limits), but you can only specify one profile at time in any given query.

1. Start with an index definition. You can add and update scoring profiles on an existing index without having to rebuild it. Use an Update Index request to post your revision.

2. Paste in the Template provided in this article.

3. Provide a name. Scoring profiles are optional, but if you add one, the name is required. Be sure to follow Cognitive Search naming conventions for fields (starts with a letter, avoids special characters and reserved words).

4. Specify boosting criteria. A single profile can contain weighted fields, functions, or both.

You should work iteratively, using a data set that will help you prove or disprove the efficacy of a given profile.

Scoring profiles can be defined in Azure portal as shown in the following screenshot, or programmatically through REST APIs or in Azure SDKs, such as the ScoringProfile class in the Azure SDK for .NET.

Using weighted fields

Use weighted fields when field context is important and queries are full text search. For example, if a query includes the term "airport", you might want "airport" in the Description field to have more weight than in the HotelName.

Weighted fields are composed of a searchable field and a positive number that is used as a multiplier. If the original field score of HotelName is 3, the boosted score for that field becomes 6, contributing to a higher overall score for the parent document itself.

JSON

```
"scoringProfiles": [
{
  "name": "boostKeywords",
  "text": {
    "weights": {
      "HotelName": 2,
      "Description": 5
    }
  }
}
```

Using functions

Use functions when simple relative weights are insufficient or don't apply, as is the case of distance and freshness, which are calculations over numeric data. You can specify multiple functions per scoring profile.

Function	Description
"freshness"	Boosts by values in a datetime field (`Edm.DateTimeOffset`). This function has a "boostingDuration" attribute so that you can specify a value representing a timespan over which boosting occurs.
"magnitude"	Boosts based on how high or low a numeric value is. Scenarios that call for this function include boosting by profit margin, highest price, lowest price, or a count of downloads. This function can only be used with `Edm.Double` and `Edm.Int` fields. For the magnitude function, you can reverse the range, high to low, if you want the inverse pattern (for example, to boost lower-priced items more than higher-priced items). Given a range of prices from $100 to $1, you would set "boostingRangeStart" at 100 and "boostingRangeEnd" at 1 to boost the lower-priced items.
"distance"	Boosts by proximity or geographic location. This function can only be used with `Edm.GeographyPoint` fields.
"tag"	Boosts by tags that are common to both search documents and query strings. Tags are provided in a "tagsParameter". This function can only be used with search fields of type `Edm.String` and `Collection(Edm.String)`.

Rules for using functions

- Functions can only be applied to fields that are attributed as filterable.
- Function type ("freshness", "magnitude", "distance", "tag") must be lower case.
- Functions can't include null or empty values.

Template

This section shows the syntax and template for scoring profiles. Refer to Property reference in the next section for descriptions of the scoring profile attributes.

JSON

```
"scoringProfiles": [
  {
    "name": "name of scoring profile",
    "text": (optional, only applies to searchable fields) {
      "weights": {
```

```
        "searchable_field_name": relative_weight_value (positive #'s),
        ...
      }
    },
    "functions": (optional) [
      {
        "type": "magnitude | freshness | distance | tag",
        "boost": # (positive number used as multiplier for raw score != 1),
        "fieldName": "(...)",
        "interpolation": "constant | linear (default) | quadratic |
logarithmic",

        "magnitude": {
          "boostingRangeStart": #,
          "boostingRangeEnd": #,
          "constantBoostBeyondRange": true | false (default)
        }

        // ( - or -)

        "freshness": {
          "boostingDuration": "..." (value representing timespan over which
boosting occurs)
        }

        // ( - or -)

        "distance": {
          "referencePointParameter": "...", (parameter to be passed in
queries to use as reference location)
          "boostingDistance": # (the distance in kilometers from the
reference location where the boosting range ends)
        }

        // ( - or -)

        "tag": {
          "tagsParameter":  "..."(parameter to be passed in queries to
specify a list of tags to compare against target field)
        }
      }
    ],
    "functionAggregation": (optional, applies only when functions are
specified) "sum (default) | average | minimum | maximum | firstMatching"
  }
],
"defaultScoringProfile": (optional) "...",
```

Property reference

Attribute	Description

Attribute	Description
name	Required. This is the name of the scoring profile. It follows the same naming conventions of a field. It must start with a letter, can't contain dots, colons or @ symbols, and can't start with the phrase azureSearch (case-sensitive).
text	Contains the weights property.
weights	Optional. Name-value pairs that specify a searchable field and a positive integer or floating-point number by which to boost a field's score. The positive integer or number becomes a multiplier for the original field score generated by the ranking algorithm. For example, if a field score is 2 and the weight value is 3, the boosted score for the field becomes 6. Individual field scores are then aggregated to create a document field score, which is then used to rank the document in the result set.
functions	Optional. A scoring function can only be applied to fields that are filterable.
functions > type	Required for scoring functions. Indicates the type of function to use. Valid values include magnitude, freshness, distance, and tag. You can include more than one function in each scoring profile. The function name must be lower case.
functions > boost	Required for scoring functions. A positive number used as multiplier for raw score. It can't be equal to 1.
functions > fieldname	Required for scoring functions. A scoring function can only be applied to fields that are part of the field collection of the index, and that are filterable. In addition, each function type introduces additional restrictions (freshness is used with datetime fields, magnitude with integer or double fields, and distance with location fields). You can only specify a single field per function definition. For example, to use magnitude twice in the same profile, you would need to include two definitions magnitude, one for each field.
functions > interpolation	Required for scoring functions. Defines the slope for which the score boosting increases from the start of the range to the end of the range. Valid values include Linear (default), Constant, Quadratic, and Logarithmic. See Set interpolations for details.

Attribute	Description
functions > magnitude	The magnitude scoring function is used to alter rankings based on the range of values for a numeric field. Some of the most common usage examples of this are: "Star ratings:" Alter the scoring based on the value within the "Star Rating" field. When two items are relevant, the item with the higher rating will be displayed first. "Margin:" When two documents are relevant, a retailer may wish to boost documents that have higher margins first. "Click counts:" For applications that track click through actions to products or pages, you could use magnitude to boost items that tend to get the most traffic. "Download counts:" For applications that track downloads, the magnitude function lets you boost items that have the most downloads.
functions > magnitude > boostingRangeStart	Sets the start value of the range over which magnitude is scored. The value must be an integer or floating-point number. For star ratings of 1 through 4, this would be 1. For margins over 50%, this would be 50.
functions > magnitude > boostingRangeEnd	Sets the end value of the range over which magnitude is scored. The value must be an integer or floating-point number. For star ratings of 1 through 4, this would be 4.
functions > magnitude > constantBoostBeyondRange	Valid values are true or false (default). When set to true, the full boost will continue to apply to documents that have a value for the target field that’s higher than the upper end of the range. If false, the boost of this function won’t be applied to documents having a value for the target field that falls outside of the range.

Attribute	Description
functions > freshness	The freshness scoring function is used to alter ranking scores for items based on values in DateTimeOffset fields. For example, an item with a more recent date can be ranked higher than older items. It's also possible to rank items like calendar events with future dates such that items closer to the present can be ranked higher than items further in the future. In the current service release, one end of the range will be fixed to the current time. The other end is a time in the past based on the boostingDuration. To boost a range of times in the future, use a negative boostingDuration. The rate at which the boosting changes from a maximum and minimum range is determined by the Interpolation applied to the scoring profile (see the figure below). To reverse the boosting factor applied, choose a boost factor of less than 1.
functions > freshness > boostingDuration	Sets an expiration period after which boosting will stop for a particular document. See Set boostingDuration in the following section for syntax and examples.
functions > distance	The distance scoring function is used to affect the score of documents based on how close or far they're relative to a reference geographic location. The reference location is given as part of the query in a parameter (using the scoringParameter query parameter) as a `lon,lat` argument.
functions > distance > referencePointParameter	A parameter to be passed in queries to use as reference location (using the scoringParameter query parameter).
functions > distance > boostingDistance	A number that indicates the distance in kilometers from the reference location where the boosting range ends.
functions > tag	The tag scoring function is used to affect the score of documents based on tags in documents and search queries. Documents that have tags in common with the search query will be boosted. The tags for the search query are provided as a scoring parameter in each search request (using the scoringParameter query parameter).

Attribute	Description
functions > tag > tagsParameter	A parameter to be passed in queries to specify tags for a particular request (using the scoringParameter query parameter). The parameter consists of a comma-delimited list of whole terms. If a given tag within the list is itself a comma-delimited list, you can use a text normalizer on the field to strip out the commas at query time (map the comma character to a space). This approach will "flatten" the list so that all terms are a single, long string of comma-delimited terms.
functionAggregation	Optional. Applies only when functions are specified. Valid values include: sum (default), average, minimum, maximum, and firstMatching. A search score is single value that is computed from multiple variables, including multiple functions. This attribute indicates how the boosts of all the functions are combined into a single aggregate boost that then is applied to the base document score. The base score is based on the tf-idf value computed from the document and the search query.
defaultScoringProfile	When executing a search request, if no scoring profile is specified, then default scoring is used (tf-idf only). You can override the built-in default, substituting a custom profile as the one to use when no specific profile is given in the search request.

Set interpolations

Interpolations allow you to set the shape of the slope used for scoring. Because scoring is high to low, the slope is always decreasing, but the interpolation determines the curve of the downward slope. The following interpolations can be used:

Interpolation	Description
`linear`	For items that are within the max and min range, the boost applied to the item will be done in a constantly decreasing amount. Linear is the default interpolation for a scoring profile.
`constant`	For items that are within the start and ending range, a constant boost will be applied to the rank results.
`quadratic`	In comparison to a Linear interpolation that has a constantly decreasing boost, Quadratic will initially decrease at smaller pace and then as it approaches the end range, it decreases at a much higher interval. This interpolation option isn't allowed in tag scoring functions.

Interpolation	Description
`logarithmic`	In comparison to a Linear interpolation that has a constantly decreasing boost, Logarithmic will initially decrease at higher pace and then as it approaches the end range, it decreases at a much smaller interval. This interpolation option isn't allowed in tag scoring functions.

Set boostingDuration

`boostingDuration` is an attribute of the `freshness` function. You use it to set an expiration period after which boosting will stop for a particular document. For example, to boost a product line or brand for a 10-day promotional period, you would specify the 10-day period as "P10D" for those documents.

`boostingDuration` must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). The pattern for this is: "P[nD][T[nH][nM][nS]]".

The following table provides several examples.

Duration	boostingDuration
1 day	"P1D"
2 days and 12 hours	"P2DT12H"
15 minutes	"PT15M"
30 days, 5 hours, 10 minutes, and 6.334 seconds	"P30DT5H10M6.334S"

For more examples, see XML Schema: Datatypes (W3.org web site) .

Extended example

The following example shows the schema of an index with two scoring profiles (`boostGenre`, `newAndHighlyRated`). Any query against this index that includes either profile as a query parameter will use the profile to score the result set.

The `boostGenre` profile uses weighted text fields, boosting matches found in albumTitle, genre, and artistName fields. The fields are boosted 1.5, 5, and 2 respectively. Why is genre boosted so much higher than the others? If search is conducted over data that is somewhat homogenous (as is the case with 'genre' in the musicstoreindex), you might need a larger variance in the relative weights. For example, in the musicstoreindex, 'rock' appears as both a genre and in identically phrased genre descriptions. If you want genre to outweigh genre description, the genre field will need a much higher relative weight.

JSON

```
{
  "name": "musicstoreindex",
  "fields": [
    { "name": "key", "type": "Edm.String", "key": true },
    { "name": "albumTitle", "type": "Edm.String" },
    { "name": "albumUrl", "type": "Edm.String", "filterable": false },
    { "name": "genre", "type": "Edm.String" },
    { "name": "genreDescription", "type": "Edm.String", "filterable": false
},
    { "name": "artistName", "type": "Edm.String" },
    { "name": "orderableOnline", "type": "Edm.Boolean" },
    { "name": "rating", "type": "Edm.Int32" },
    { "name": "tags", "type": "Collection(Edm.String)" },
    { "name": "price", "type": "Edm.Double", "filterable": false },
    { "name": "margin", "type": "Edm.Int32", "retrievable": false },
    { "name": "inventory", "type": "Edm.Int32" },
    { "name": "lastUpdated", "type": "Edm.DateTimeOffset" }
  ],
  "scoringProfiles": [
    {
      "name": "boostGenre",
      "text": {
        "weights": {
          "albumTitle": 1.5,
          "genre": 5,
          "artistName": 2
        }
      }
    },
    {
      "name": "newAndHighlyRated",
      "functions": [
        {
          "type": "freshness",
          "fieldName": "lastUpdated",
          "boost": 10,
          "interpolation": "quadratic",
```

```
          "freshness": {
            "boostingDuration": "P365D"
          }
        },
        {
          "type": "magnitude",
          "fieldName": "rating",
          "boost": 10,
          "interpolation": "linear",
          "magnitude": {
            "boostingRangeStart": 1,
            "boostingRangeEnd": 5,
            "constantBoostBeyondRange": false
          }
        }
      ]
    }
  ],
  "suggesters": [
    {
      "name": "sg",
      "searchMode": "analyzingInfixMatching",
      "sourceFields": [ "albumTitle", "artistName" ]
    }
  ]
}
```

See also

- Relevance and scoring in Azure Cognitive Search
- REST API Reference
- Create Index API
- Azure Cognitive Search .NET SDK
- What Are Scoring Profiles?

Configure semantic ranking and return captions in search results

Article • 03/24/2023 • 14 minutes to read

> **Important**
>
> Semantic search is in public preview under **supplemental terms of use**. It's available through Azure portal, preview REST APIs, and beta SDKs. This feature is billable. See **Availability and pricing**.

In this article, you'll learn how to invoke a semantic ranking algorithm over a result set, promoting the most semantically relevant results to the top of the stack. You can also get semantic captions, with highlights over the most relevant terms and phrases, and semantic answers.

There are two main activities to perform:

- Add a semantic configuration to an index
- Add parameters to a query request

Prerequisites

- A search service on Standard tier (S1, S2, S3) or Storage Optimized tier (L1, L2), in these regions: Australia East, East US, East US 2, North Central US, South Central US, West US, West US 2, North Europe, UK South, West Europe.

  If you have an existing S1 or greater service in one of these regions, you can enable semantic search without having to create a new service.

- Semantic search enabled on your search service.

- An existing search index with rich content in a supported query language. Semantic search works best on content that is informational or descriptive.

- Review the Semantic search overview if you need an introduction to the feature.

> **Note**
>
> Captions and answers are extracted verbatim from text in the search document. The semantic subsystem determines what part of your content has the characteristics of

a caption or answer, but it doesn't compose new sentences or phrases. For this reason, content that includes explanations or definitions work best for semantic search.

1 - Choose a client

You'll need a search client that supports preview APIs on the query request. Here are some options:

- Search explorer in Azure portal, recommended for initial exploration.
- Postman app using the 2021-04-30-Preview REST APIs. See this Quickstart for help with setting up your requests.
- Azure.Search.Documents 11.4.0-beta.5 in the Azure SDK for .NET Preview.
- Azure.Search.Documents 11.3.0b6 in the Azure SDK for Python.

2 - Create a semantic configuration

> **Important**
>
> A semantic configuration is required for the 2021-04-30-Preview REST APIs, Search explorer, and some versions of the beta SDKs. If you're using the 2020-06-30-preview REST API, skip this step and use the **"searchFields" approach for field prioritization** instead.

A *semantic configuration* specifies how fields are used in semantic ranking. It gives the underlying models hints about which index fields are most important for semantic ranking, captions, highlights, and answers.

You'll add a semantic configuration to your index definition. The tabbed sections below provide instructions for the REST APIs, Azure portal, and the .NET SDK Preview.

You can add or update a semantic configuration at any time without rebuilding your index. When you issue a query, you'll add the semantic configuration (one per query) that specifies which semantic configuration to use for the query.

1. Review the properties you'll need to specify. A semantic configuration has a name and at least one each of the following properties:

- **Title field** - A title field should be a concise description of the document, ideally a string that is under 25 words. This field could be the title of the document, name of the product, or item in your search index. If you don't have a title in your search index, leave this field blank.
- **Content fields** - Content fields should contain text in natural language form. Common examples of content are the body of a document, the description of a product, or other free-form text.
- **Keyword fields** - Keyword fields should be a list of keywords, such as the tags on a document, or a descriptive term, such as the category of an item.

You can only specify one title field but you can specify as many content and keyword fields as you like. For content and keyword fields, list the fields in priority order because lower priority fields may get truncated.

2. For the above properties, determine which fields to assign.

   A field must be a supported data type and it should contain strings. If you happen to include an invalid field, there's no error, but those fields won't be used in semantic ranking.

Data type	Example from hotels-sample-index
Edm.String	HotelName, Category, Description
Edm.ComplexType	Address.StreetNumber, Address.City, Address.StateProvince, Address.PostalCode
Collection(Edm.String)	Tags (a comma-delimited list of strings)

> **Note**
>
> Subfields of Collection(Edm.ComplexType) fields aren't currently supported by semantic search and won't be used for semantic ranking, captions, or answers.

Azure portal

1. Sign in to Azure portal and navigate to a search service that has semantic search enabled.

2. Open an index.

3. Select **Semantic Configurations** and then select **Add Semantic Configuration**.

The **New Semantic Configuration** page opens with options for selecting a title field, content fields, and keyword fields. Make sure to list content fields and keyword fields in priority order.

Select **OK** to save the changes.

Tip

To see an example of creating a semantic configuration and using it to issue a semantic query, check out the **semantic search Postman sample**.

2b - Use searchFields for field prioritization

This step is only for solutions using the 2020-06-30-Preview REST API or a beta SDK that doesn't support semantic configurations. Instead of setting field prioritization in the index through a semantic configuration, you'll set the priority at query time, using the "searchFields" parameter of a query.

Using "searchFields" for field prioritization was an early implementation detail that won't be supported once semantic search exits public preview. We encourage you to use semantic configurations if your application requirements allow it.

HTTP

```
POST https://[service name].search.windows.net/indexes/[index
name]/docs/search?api-version=2020-06-30-Preview
{
    "search": " Where was Alan Turing born?",
    "queryType": "semantic",
    "searchFields": "title,url,body",
```

```
    "queryLanguage": "en-us"
}
```

Field order is critical because the semantic ranker limits the amount of content it can process while still delivering a reasonable response time. Content from fields at the start of the list are more likely to be included; content from the end could be truncated if the maximum limit is reached. For more information, see Pre-processing during semantic ranking.

- If you're specifying just one field, choose a descriptive field where the answer to semantic queries might be found, such as the main content of a document.
- For two or more fields in searchFields:
  - The first field should always be concise (such as a title or name), ideally a string that is under 25 words.
  - If the index has a URL field that is human readable such as `www.domain.com/name-of-the-document-and-other-details` (rather than machine focused, such as `www.domain.com/?id=23463&param=eis`), place it second in the list (or first if there's no concise title field).
  - Follow the above fields with other descriptive fields, where the answer to semantic queries may be found, such as the main content of a document.

When setting "searchFields", choose only fields of the following supported data types:

Data type	Example from hotels-sample-index
Edm.String	HotelName, Category, Description
Edm.ComplexType	Address.StreetNumber, Address.City, Address.StateProvince, Address.PostalCode
Collection(Edm.String)	Tags (a comma-delimited list of strings)

If you happen to include an invalid field, there's no error, but those fields won't be used in semantic ranking.

3 - Avoid features that bypass relevance scoring

Several query capabilities in Cognitive Search don't undergo relevance scoring, and some bypass the full text search engine altogether. If your query logic includes the

following features, you won't get relevance scores or semantic ranking on your results:

- Filters, fuzzy search queries, and regular expressions iterate over untokenized text, scanning for verbatim matches in the content. Search scores for all of the above query forms are a uniform 1.0, and won't provide meaningful input for semantic ranking.
- Sorting (orderBy clauses) on specific fields will also override search scores and semantic score. Given that semantic score is used to order results, including explicit sort logic will cause an HTTP 400 error to be returned.

4 - Set up the query

Your next step is adding parameters to the query request. To be successful, your query should be full text search (using the "search" parameter to pass in a string), and the index should contain text fields with rich semantic content.

Azure portal

Search explorer has been updated to include options for semantic queries. To configure semantic ranking in the portal, follow the steps below:

1. Open the Azure portal and navigate to a search service that has semantic search enabled.
2. Select **Search explorer** at the top of the overview page.
3. Choose an index that has content in a supported language.
4. In Search explorer, set query options that enable semantic queries, semantic configurations, and spell correction. You can also paste the required query parameters into the query string.

5 - Evaluate the response

Only the top 50 matches from the initial results can be semantically ranked. As with all queries, a response is composed of all fields marked as retrievable, or just those fields listed in the select parameter. A response includes the original relevance score, and might also include a count, or batched results, depending on how you formulated the request.

In semantic search, the response has more elements: a new semantically ranked relevance score, an optional caption in plain text and with highlights, and an optional answer. If your results don't include these extra elements, then your query might be misconfigured. As a first step towards troubleshooting the problem, check the semantic configuration to ensure it's specified in both the index definition and query.

In a client app, you can structure the search page to include a caption as the description of the match, rather than the entire contents of a specific field. This approach is useful when individual fields are too dense for the search results page.

The response for the above example query returns the following match as the top pick. Captions are returned because the "captions" property is set, with plain text and highlighted versions. Answers are omitted from the example because one couldn't be determined for this particular query and corpus.

JSON

```
"@odata.count": 35,
"@search.answers": [],
"value": [
    {
        "@search.score": 1.8810667,
        "@search.rerankerScore": 1.1446577133610845,
```

```
        "@search.captions": [
            {
                "text": "Oceanside Resort. Luxury. New Luxury Hotel. Be the
first to stay. Bay views from every room, location near the pier, rooftop
pool, waterfront dining & more.",
                "highlights": "<strong>Oceanside Resort.</strong> Luxury.
New Luxury Hotel. Be the first to stay.<strong> Bay</strong> views from
every room, location near the pier, rooftop pool, waterfront dining & more."
            }
        ],
        "HotelName": "Oceanside Resort",
        "Description": "New Luxury Hotel. Be the first to stay. Bay views
from every room, location near the pier, rooftop pool, waterfront dining &
more.",
        "Category": "Luxury"
    },
```

Next steps

Recall that semantic ranking and responses are built over an initial result set. Any logic that improves the quality of the initial results will carry forward to semantic search. As a next step, review the features that contribute to initial results, including analyzers that affect how strings are tokenized, scoring profiles that can tune results, and the default relevance algorithm.

- Analyzers for text processing
- Similarity ranking algorithm
- Scoring profiles
- Semantic search overview
- Semantic ranking algorithm

Analyze performance in Azure Cognitive Search

Article • 03/01/2023 • 8 minutes to read

This article describes the tools, behaviors, and approaches for analyzing query and indexing performance in Cognitive Search.

Develop baseline numbers

In any large implementation, it's critical to do a performance benchmarking test of your Cognitive Search service before you roll it into production. You should test both the search query load that you expect, but also the expected data ingestion workloads (if possible, run both workloads simultaneously). Having benchmark numbers helps to validate the proper search tier, service configuration, and expected query latency.

To develop benchmarks, we recommend the azure-search-performance-testing (GitHub) tool.

To isolate the effects of a distributed service architecture, try testing on service configurations of one replica and one partition.

> **Note**
>
> For the Storage Optimized tiers (L1 and L2), you should expect a lower query throughput and higher latency than the Standard tiers.

Use resource logging

The most important diagnostic tool at an administrator's disposal is resource logging. Resource logging is the collection of operational data and metrics about your search service. Resource logging is enabled through Azure Monitor. There are costs associated with using Azure Monitor and storing data, but if you enable it for your service, it can be instrumental in investigating performance issues.

The following image shows the chain of events in a query request and response. Latency can occur at any one of them, whether during a network transfer, processing of content in the app services layer, or on a search service. A key benefit of resource logging is that activities are logged from the search service perspective, which means that the log can

help you determine if the performance issue is due to problems with the query or indexing, or some other point of failure.

Resource logging gives you options for storing logged information. We recommend using Log Analytics so that you can execute advanced Kusto queries against the data to answer many questions about usage and performance.

On your search service portal pages, you can enable logging through **Diagnostic settings**, and then issue Kusto queries against Log Analytics by choosing **Logs**. For more information about setting up, see Collect and analyze log data.

Throttling behaviors

Throttling occurs when the search service is at capacity. Throttling can occur during queries or indexing. From the client side, an API call results in a 503 HTTP response when it has been throttled. During indexing, there's also the possibility of receiving a 207 HTTP response, which indicates that one or more items failed to index. This error is an indicator that the search service is getting close to capacity.

As a rule of thumb, try to quantify the amount of throttling and any patterns. For example, if one search query out of 500,000 is throttled, it might not be worth investigating. However, if a large percentage of queries is throttled over a period, this would be a greater concern. By looking at throttling over a period, it also helps to identify time frames where throttling might more likely occur and help you decide how to best accommodate that.

A simple fix to most throttling issues is to throw more resources at the search service (typically replicas for query-based throttling, or partitions for indexing-based throttling). However, increasing replicas or partitions adds cost, which is why it's important to know the reason why throttling is occurring at all. Investigating the conditions that cause throttling will be explained in the next several sections.

Below is an example of a Kusto query that can identify the breakdown of HTTP responses from the search service that has been under load. Over a 7-day period, the rendered bar chart shows that a relatively large percentage of the search queries were throttled, in comparison to the number of successful (200) responses.

Kusto

```
AzureDiagnostics
| where TimeGenerated > ago(7d)
| summarize count() by resultSignature_d
| render barchart
```

Examining throttling over a specific time period can help you identify the times where throttling might occur more frequently. In the below example, a time series chart is used to show the number of throttled queries that occurred over a specified time frame. In this case, the throttled queries correlated with the times in with the performance benchmarking was performed.

Kusto

```
let ['_startTime']=datetime('2021-02-25T20:45:07Z');
let ['_endTime']=datetime('2021-03-03T20:45:07Z');
let intervalsize = 1m;
AzureDiagnostics
| where TimeGenerated > ago(7d)
```

```
| where resultSignature_d != 403 and resultSignature_d != 404 and
OperationName in ("Query.Search", "Query.Suggest", "Query.Lookup",
"Query.Autocomplete")
| summarize
  ThrottledQueriesPerMinute=bin(countif(OperationName in ("Query.Search",
"Query.Suggest", "Query.Lookup", "Query.Autocomplete") and resultSignature_d
== 503)/(intervalsize/1m), 0.01)
  by bin(TimeGenerated, intervalsize)
| render timechart
```

Measure individual queries

In some cases, it can be useful to test individual queries to see how they're performing. To do this, it's important to be able to see how long the search service takes to complete the work, as well as how long it takes to make the round-trip request from the client and back to the client. The diagnostics logs could be used to look up individual operations, but it might be easier to do this all from a client tool, such as Postman.

In the example below, a REST-based search query was executed. Cognitive Search includes in every response the number of milliseconds it takes to complete the query, visible in the Headers tab, in "elapsed-time". Next to Status at the top of the response, you'll find the round-trip duration, in this case, 418 milliseconds (ms). In the results section, the “Headers” tab was chosen. Using these two values, highlighted with a red box in the image below, we see the search service took 21 ms to complete the search query and the entire client round-trip request took 125 ms. By subtracting these two numbers we can determine that it took 104-ms additional time to transmit the search query to the search service and to transfer the search results back to the client.

This technique helps you isolate network latencies from other factors impacting query performance.

Query rates

One potential reason for your search service to throttle requests is due to the sheer number of queries being performed where volume is captured as queries per second (QPS) or queries per minute (QPM). As your search service receives more QPS, it will typically take longer and longer to respond to those queries until it can no longer keep up, as which it will send back a throttling 503 HTTP response.

The following Kusto query shows query volume as measured in QPM, along with average duration of a query in milliseconds (AvgDurationMS) and the average number of documents (AvgDocCountReturned) returned in each one.

Kusto

```
AzureDiagnostics
| where OperationName == "Query.Search" and TimeGenerated > ago(1d)
| extend MinuteOfDay = substring(TimeGenerated, 0, 16)
| project MinuteOfDay, DurationMs, Documents_d, IndexName_s
| summarize QPM=count(), AvgDuractionMs=avg(DurationMs),
AvgDocCountReturned=avg(Documents_d)  by MinuteOfDay
| order by MinuteOfDay desc
| render timechart
```

> **Tip**
>
> To reveal the data behind this chart, remove the line `| render timechart` and then rerun the query.

Impact of indexing on queries

An important factor to consider when looking at performance is that indexing uses the same resources as search queries. If you're indexing a large amount of content, you can expect to see latency grow as the service tries to accommodate both workloads.

If queries are slowing down, look at the timing of indexing activity to see if it coincides with query degradation. For example, perhaps an indexer is running a daily or hourly job that correlates with the decreased performance of the search queries.

This section provides a set of queries that can help you visualize the search and indexing rates. For these examples, the time range is set in the query. Be sure to indicate **Set in query** when running the queries in Azure portal.

Average Query Latency

In the below query, an interval size of 1 minute is used to show the average latency of the search queries. From the chart, we can see that the average latency was low until 5:45pm and lasted until 5:53pm.

Kusto

```
let intervalsize = 1m;
let _startTime = datetime('2021-02-23 17:40');
let _endTime = datetime('2021-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between(['_startTime']..['_endTime']) // Time range
filtering
| summarize AverageQueryLatency = avgif(DurationMs, OperationName in
("Query.Search", "Query.Suggest", "Query.Lookup", "Query.Autocomplete"))
    by bin(TimeGenerated, intervalsize)
| render timechart
```

Average Queries Per Minute (QPM)

The following query looks at the average number of queries per minute to ensure that there wasn't a spike in search requests that might have affected the latency. From the chart, we can see there's some variance, but nothing to indicate a spike in request count.

Kusto

```
let intervalsize = 1m;
let _startTime = datetime('2021-02-23 17:40');
let _endTime = datetime('2021-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between(['_startTime'] .. ['_endTime']) // Time range
filtering
| summarize QueriesPerMinute=bin(countif(OperationName in ("Query.Search",
"Query.Suggest", "Query.Lookup", "Query.Autocomplete"))/(intervalsize/1m),
0.01)
  by bin(TimeGenerated, intervalsize)
| render timechart
```

Indexing Operations Per Minute (OPM)

Here we'll look at the number of Indexing operations per minute. From the chart, we can see that a large amount of data was indexed started at 5:42 pm and ended at 5:50pm. This indexing began 3 minutes before the search queries started becoming latent and ended 3 minutes before the search queries were no longer latent.

From this insight, we can see that it took about 3 minutes for the search service to become busy enough for indexing to affect query latency. We can also see that after indexing completed, it took another 3 minutes for the search service to complete all the work from the newly indexed content, and for query latency to resolve.

Kusto

```
let intervalsize = 1m;
let _startTime = datetime('2021-02-23 17:40');
let _endTime = datetime('2021-02-23 18:00');
AzureDiagnostics
| where TimeGenerated between(['_startTime'] .. ['_endTime']) // Time range
filtering
| summarize IndexingOperationsPerSecond=bin(countif(OperationName ==
"Indexing.Index")/ (intervalsize/1m), 0.01)
  by bin(TimeGenerated, intervalsize)
| render timechart
```

Background service processing

It isn't unusual to see periodic spikes in query or indexing latency. Spikes might occur in response to indexing or high query rates, but could also occur during merge operations. Search indexes are stored in chunks - or shards. Periodically, the system merges smaller shards into large shards, which can help optimize service performance. This merge process also cleans up documents that have previously been marked for deletion from the index, resulting in the recovery of storage space.

Merging shards is fast, but also resource intensive and thus has the potential to degrade service performance. If you notice short bursts of query latency, and those bursts coincide with recent changes to indexed content, you can assume the latency is due to shard merge operations.

Next steps

Review these articles related to analyzing service performance.

- Performance tips
- Choose a service tier

- Manage capacity

Additional resources

Documentation

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Security filters for trimming results - Azure Cognitive Search

Learn how to implement security privileges at the document level for Azure Cognitive Search search results, using security filters and user identities.

Performance benchmarks - Azure Cognitive Search

Learn about the performance of Azure Cognitive Search through various performance benchmarks

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Quickstart: Text translation and entity recognition - Azure Cognitive Search

Use the Import Data wizard and AI cognitive skills to detect language, translate text, and recognize entities. The new fields created through AI become searchable text in an Azure Cognitive Search index.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Monitor queries - Azure Cognitive Search

Monitor query metrics for performance and throughput. Collect and analyze query string inputs in resource logs.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Show 5 more

Azure Cognitive Search performance benchmarks

Article • 01/31/2023 • 8 minutes to read

Azure Cognitive Search's performance depends on a variety of factors including the size of your search service and the types of queries you're sending. To help estimate the size of search service needed for your workload, we've run several benchmarks to document the performance for different search services and configurations. These benchmarks in no way guarantee a certain level of performance from your service but can give you an idea of the performance you can expect.

To cover a range of different use cases, we ran benchmarks for two main scenarios:

- **E-commerce search** - This benchmark emulates a real e-commerce scenario and is based on the Nordic e-commerce company CDON.
- **Document search** - This scenario is comprised of keyword search over full text documents from Semantic Scholar. This emulates a typical document search solution.

While these scenarios reflect different use cases, every scenario is different so we always recommend performance testing your individual workload. We've published a performance testing solution using JMeter so you can run similar tests against your own service.

Testing methodology

To benchmark Azure Cognitive Search's performance, we ran tests for two different scenarios at different tiers and replica/partition combinations.

To create these benchmarks, the following methodology was used:

1. The test begins at `x` queries per second (QPS) for 180 seconds. This was usually 5 or 10 QPS.
2. QPS then increased by `x` and ran for another 180 seconds
3. Every 180 seconds, the test increased by `x` QPS until average latency increased above 1000 ms or less than 99% of queries succeeded.

The following graph gives a visual example of what the test's query load looks like:

Each scenario used at least 10,000 unique queries to avoid tests being overly skewed by caching.

> ⓘ **Important**
>
> These tests only include query workloads. If you expect to have a high volume of indexing operations, be sure to factor that into your estimation and performance testing. Sample code for simulating indexing can be found in this **tutorial**.

Definitions

- **Maximum QPS** - the maximum QPS numbers are based on the highest QPS achieved in a test where 99% of queries completed successfully without throttling and average latency stayed under 1000 ms.
- **Percentage of max QPS** - A percentage of the maximum QPS achieved for a particular test. For example, if a given test reached a maximum of 100 QPS, 20% of max QPS would be 20 QPS.
- **Latency** - The server's latency for a query; these numbers don't include round trip delay (RTT). Values are in milliseconds (ms).

Disclaimer

The code we used to run these benchmarks is available on the azure-search-performance-testing repository. It's worth noting that we observed slightly lower QPS

levels with the JMeter performance testing solution than in the benchmarks. The differences can be attributed to differences in the style of the tests. This speaks to the importance of making your performance tests as similar to your production workload as possible.

These benchmarks in no way guarantee a certain level of performance from your service but can give you an idea of the performance you can expect based on your scenario.

If you have any questions or concerns, reach out to us at azuresearch_contact@microsoft.com.

Benchmark 1: E-commerce search

CDON

This benchmark was created in partnership with the e-commerce company, CDON, the Nordic region's largest online marketplace with operations in Sweden, Finland, Norway, and Denmark. Through its 1,500 merchants, CDON offers a wide range assortment that includes over 8 million products. In 2020, CDON had over 120 million visitors and 2 million active customers. You can learn more about CDON's use of Azure Cognitive Search in this article.

To run these tests, we used a snapshot of CDON's production search index and thousands of unique queries from their website.

Scenario Details

- **Document Count**: 6,000,000
- **Index Size**: 20 GB
- **Index Schema**: a wide index with 250 fields total, 25 searchable fields, and 200 facetable/filterable fields
- **Query Types**: full text search queries including facets, filters, ordering, and scoring profiles

S1 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	104 ms	35 ms	115 ms	177 ms	257 ms	738 ms
50%	140 ms	47 ms	144 ms	241 ms	400 ms	1175 ms
80%	239 ms	77 ms	248 ms	466 ms	763 ms	1752 ms

S2 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	56 ms	21 ms	68 ms	106 ms	132 ms	210 ms
50%	71 ms	26 ms	83 ms	132 ms	177 ms	329 ms
80%	140 ms	47 ms	153 ms	293 ms	452 ms	924 ms

S3 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

In this case, we see that adding a second partition significantly increases the maximum QPS but adding a third partition provides diminishing marginal returns. The smaller improvement is likely because all of the data is already being pulled into the S3's active memory with just two partitions.

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	50 ms	20 ms	64 ms	83 ms	98 ms	160 ms
50%	62 ms	24 ms	80 ms	107 ms	130 ms	253 ms
80%	115 ms	38 ms	121 ms	218 ms	352 ms	828 ms

Benchmark 2: Document search

Scenario Details

- **Document Count**: 7.5 million
- **Index Size**: 22 GB
- **Index Schema**: 23 fields; 8 searchable, 10 filterable/facetable
- **Query Types**: keyword searches with facets and hit highlighting

S1 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	67 ms	44 ms	77 ms	103 ms	126 ms	216 ms
50%	93 ms	59 ms	110 ms	150 ms	184 ms	304 ms
80%	150 ms	96 ms	184 ms	248 ms	297 ms	424 ms

S2 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	45 ms	31 ms	55 ms	73 ms	84 ms	109 ms
50%	63 ms	39 ms	81 ms	106 ms	123 ms	163 ms
80%	115 ms	73 ms	145 ms	191 ms	224 ms	291 ms

S3 Performance

Queries per second

The following chart shows the highest query load a service could handle for an extended period of time in terms of queries per second (QPS).

Query latency

Query latency varies based on the load of the service and services under higher stress will have a higher average query latency. The following table shows the 25th, 50th, 75th, 90th, 95th, and 99th percentiles of query latency for three different usage levels.

Percentage of max QPS	Average latency	25%	75%	90%	95%	99%
20%	43 ms	29 ms	53 ms	74 ms	86 ms	111 ms
50%	65 ms	37 ms	85 ms	111 ms	128 ms	164 ms
80%	126 ms	83 ms	162 ms	205 ms	233 ms	281 ms

Takeaways

Through these benchmarks, you can get an idea of the performance Azure Cognitive Search offers. You can also see difference between services at different tiers.

Some key take ways from these benchmarks are:

- An S2 can typically handle at least four times the query volume as an S1
- An S2 will typically have lower latency than an S1 at comparable query volumes
- As you add replicas, the QPS a service can handle typically scales linearly (for example, if one replica can handle 10 QPS then five replicas can usually handle 50 QPS)
- The higher the load on the service, the higher the average latency will be

You can also see that performance can vary drastically between scenarios. If you're not getting the performance you expect, check out the tips for better performance.

Next steps

Now that you've seen the performance benchmarks, you can learn more about how to analyze Cognitive Search's performance and key factors that influence performance.

Analyze performance | Tips for better performance

Additional resources

Documentation

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Index large data sets for full text search - Azure Cognitive Search

Strategies for large data indexing or computationally intensive indexing through batch mode, resourcing, and scheduled, parallel, and distributed indexing.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Analyze performance - Azure Cognitive Search

TBD

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Feature descriptions - Azure Cognitive Search

Explore the feature categories of Azure Cognitive Search.

Create a search index - Azure Cognitive Search

Create a search index using the Azure portal, REST APIs, or an Azure SDK.

Schedule indexer execution - Azure Cognitive Search

Learn how to schedule Azure Cognitive Search indexers to index content at specific intervals, or at specific dates and times.

Show 5 more

Tips for better performance in Azure Cognitive Search

Article • 04/21/2023

This article is a collection of tips and best practices that are often recommended for boosting performance. Knowing which factors are most likely to impact search performance can help you avoid inefficiencies and get the most out of your search service. Some key factors include:

- Index composition (schema and size)
- Query design
- Service capacity (tier, and the number of replicas and partitions)

> ⓘ **Note**
>
> Looking for strategies on high volume indexing? See **Index large data sets in Azure Cognitive Search**.

Index size and schema

Queries run faster on smaller indexes. This is partly a function of having fewer fields to scan, but it's also due to how the system caches content for future queries. After the first query, some content remains in memory where it's searched more efficiently. Because index size tends to grow over time, one best practice is to periodically revisit index composition, both schema and documents, to look for content reduction opportunities. However, if the index is right-sized, the only other calibration you can make is to increase capacity: either by adding replicas or upgrading the service tier. The section "Tip: Upgrade to a Standard S2 tier" discusses the scale up versus scale out decision.

Schema complexity can also adversely affect indexing and query performance. Excessive field attribution builds in limitations and processing requirements. Complex types take longer to index and query. The next few sections explore each condition.

Tip: Be selective in field attribution

A common mistake that administrators and developers make when creating a search index is selecting all available properties for the fields, as opposed to only selecting just the properties that are needed. For example, if a field doesn't need to be full text searchable, skip that field when setting the searchable attribute.

Support for filters, facets, and sorting can quadruple storage requirements. If you add suggesters, storage requirements go up even more. For an illustration on the impact of attributes on storage, see Attributes and index size.

Summarized, the ramifications of over-attribution include:

- Degradation of indexing performance due to the extra work required to process the content in the field, and then store it within the search inverted index (set the "searchable" attribute only on fields that contain searchable content).
- Creates a larger surface that each query has to cover. All fields marked as searchable are scanned in a full text search.
- Increases operational costs due to extra storage. Filtering and sorting requires additional space for storing original (non-analyzed) strings. Avoid setting filterable or sortable on fields that don't need it.
- In many cases, over attribution limits the capabilities of the field. For example, if a field is facetable, filterable, and searchable, you can only store 16 KB of text within a field, whereas a searchable field can hold up to 16 MB of text.

> **Note**
>
> Only unnecessary attribution should be avoided. Filters and facets are often essential to the search experience, and in cases where filters are used, you frequently need sorting so that you can order the results (filters by themselves return in an unordered set).

Tip: Consider alternatives to complex types

Complex data types are useful when data has a complicated nested structure, such as the parent-child elements found in JSON documents. The downside of complex types is

the extra storage requirements and additional resources required to index the content, in comparison to non-complex data types.

In some cases, you can avoid these tradeoffs by mapping a complex data structure to a simpler field type, such as a Collection. Alternatively, you might opt for flattening a field hierarchy into individual root-level fields.

Query design

Query composition and complexity are one of the most important factors for performance, and query optimization can drastically improve performance. When designing queries, think about the following points:

- **Number of searchable fields.** Each additional searchable field results in more work for the search service. You can limit the fields being searched at query time using the "searchFields" parameter. It's best to specify only the fields that you care about to improve performance.

- **Amount of data being returned.** Retrieving a large amount content can make queries slower. When structuring a query, return only those fields that you need to render the results page, and then retrieve remaining fields using the Lookup API once a user selects a match.

- **Use of partial term searches.** Partial term searches, such as prefix search, fuzzy search, and regular expression search, are more computationally expensive than typical keyword searches, as they require full index scans to produce results.

- **Number of facets.** Adding facets to queries requires aggregations for each query. Requesting a higher "count" for a facet also requires extra work by the service. In general, only add the facets that you plan to render in your app and avoid requesting a high count for facets unless necessary.

- **High skip values.** Setting the $skip parameter to a high value (for example, in the thousands) increases search latency because the engine is retrieving and ranking a larger volume of documents for each request. For performance reasons, it's best to avoid high $skip values and use other techniques instead, such as filtering, to retrieve large numbers of documents.

- **Limit high cardinality fields.** A *high cardinality field* refers to a facetable or filterable field that has a significant number of unique values, and as a result, consumes significant resources when computing results. For example, setting a Product ID or Description field as facetable and filterable would count as high cardinality because most of the values from document to document are unique.

Tip: Use search functions instead overloading filter criteria

As a query uses increasingly complex filter criteria, the performance of the search query will degrade. Consider the following example that demonstrates the use of filters to trim results based on a user identity:

JSON

```
$filter= userid eq 123 or userid eq 234 or userid eq 345 or userid eq 456 or
userid eq 567
```

In this case, the filter expressions are used to check whether a single field in each document is equal to one of many possible values of a user identity. You're most likely to find this pattern in applications that implement security trimming (checking a field containing one or more principal IDs against a list of principal IDs representing the user issuing the query).

A more efficient way to execute filters that contain a large number of values is to use search.in function, as shown in this example:

JSON

```
search.in(userid, '123,234,345,456,567', ',')
```

Tip: Add partitions for slow individual queries

When query performance is slowing down in general, adding more replicas frequently solves the issue. But what if the problem is a single query that takes too long to complete? In this scenario, adding replicas won't help, but more partitions might. A partition splits data across extra computing resources. Two partitions split data in half, a third partition splits it into thirds, and so forth.

One positive side-effect of adding partitions is that slower queries sometimes perform faster due to parallel computing. We've noted parallelization on low selectivity queries, such as queries that match many documents, or facets providing counts over a large

number of documents. Since significant computation is required to score the relevancy of the documents, or to count the numbers of documents, adding extra partitions helps queries complete faster.

To add partitions, use Azure portal, PowerShell, Azure CLI, or a management SDK.

Service capacity

A service is overburdened when queries take too long or when the service starts dropping requests. If this happens, you can address the problem by upgrading the service or by adding capacity.

The tier of your search service and the number of replicas/partitions also have a large impact on performance. Each progressively higher tier provides faster CPUs and more memory, both of which have a positive impact on performance.

Tip: Upgrade to a Standard S2 tier

The Standard S1 search tier is often where customers start. A common pattern for S1 services is that indexes grow over time, which requires more partitions. More partitions lead to slower response times, so more replicas are added to handle the query load. As you can imagine, the cost of running an S1 service has now progressed to levels beyond the initial configuration.

At this juncture, an important question to ask is whether it would be beneficial to move to a higher tier, as opposed to progressively increasing the number of partitions or replicas of the current service.

Consider the following topology as an example of a service that has taken on increasing levels of capacity:

- Standard S1 tier
- Index Size: 190 GB
- Partition Count: 8 (on S1, partition size is 25 GB per partition)
- Replica Count: 2
- Total Search Units: 16 (8 partitions x 2 replicas)
- Hypothetical Retail Price: ~$4,000 USD / month (assume $250 USD x 16 search units)

Suppose the service administrator is still seeing higher latency rates and is considering adding another replica. This would change the replica count from 2 to 3 and as a result change the Search Unit count to 24 and a resulting price of $6,000 USD/month.

However, if the administrator chose to move to a Standard S2 tier the topology would look like:

- Standard S2 tier
- Index Size: 190 GB
- Partition Count: 2 (on S2, partition size is 100 GB per partition)
- Replica Count: 2
- Total Search Units: 4 (2 partitions x 2 replicas)
- Hypothetical Retail Price: ~$4,000 USD / month ($1000 USD x 4 search units)

As this hypothetical scenario illustrates, you can have configurations on lower tiers that result in similar costs as if you had opted for a higher tier in the first place. However, higher tiers come with premium storage, which makes indexing faster. Higher tiers also have much more compute power, as well as extra memory. For the same costs, you could have more powerful infrastructure backing the same index.

An important benefit of added memory is that more of the index can be cached, resulting in lower search latency, and a greater number of queries per second. With this extra power, the administrator may not need to even need to increase the replica count and could potentially pay less than by staying on the S1 service.

Tip: Consider alternatives to regular expression queries

Regular expression queries or regex can be particularly expensive. While they can be very useful for complex searches, they also may require a lot of processing power to be executed, especially if the regular expression has a lot of complexity or when searching through a large amount of data. This would result in high search latency. In order to reduce the search latency, try to simplify the regular expression or break the complex query down into smaller, more manageable queries.

Next steps

Review these other articles related to service performance:

- Analyze performance
- Index large data sets in Azure Cognitive Search
- Choose a service tier
- Plan or add capacity
- Case Study: Use Cognitive Search to Support Complex AI Scenarios

Monitoring Azure Cognitive Search

Article • 03/10/2023 • 7 minutes to read

Azure Monitor is enabled with every subscription to provide monitoring capabilities over all Azure resources, including Cognitive Search. When you sign up for search, Azure Monitor collects **activity logs** and **platform metrics** as soon as you start using the service.

Optionally, you can enable diagnostic settings to collect **resource logs**. Resource logs contain detailed information about search service operations that's useful for deeper analysis and investigation.

This article explains how monitoring works for Azure Cognitive Search. It also describes the system APIs that return information about your service.

> **Note**
>
> Cognitive Search doesn't monitor individual user access to content on the search service. If you require this level of monitoring, you'll need to implement it in your client application.

Monitoring in Azure portal

In the search service pages in Azure portal, you can find the current status of operations and capacity.

- **Monitoring** tab in the **Overview** page summarizes key query metrics, including search latency, search queries per second, and throttled queries. On the next tab over (not shown), **Usage** shows available capacity and the quantity of indexes, indexers, data sources, and skillsets relative to the maximum allowed for your service tier.

- **Activity log** on the navigation menu captures service-level events: service creation, configuration, and deletion.

- Further down the navigation menu, the **Monitoring** section includes actions for Azure Monitor, filtered for search. Here, you can enable diagnostic settings and resource logging, and specify how you want the data stored.

> **Note**
>
> Because portal pages are refreshed every few minutes, the numbers reported are approximate, intended to give you a general sense of how well your system is handling requests. Actual metrics, such as queries per second (QPS) may be higher or lower than the number shown on the page. If precision is a requirement, consider using APIs.

Get system data from REST APIs

Cognitive Search REST APIs provide the **Usage** data that's visible in the portal. This information is retrieved from your search service, which you can obtain programmatically:

- Service Statistics (REST)
- Index Statistics (REST)
- Document Counts (REST)
- Indexer Status (REST)

For REST calls, use an admin API key and Postman or another REST client to query your search service.

Monitor activity logs

In Azure Cognitive Search, **activity logs** reflect control plane activity, such as service creation and configuration, or API key usage or management.

Activity logs are collected free of charge, with no configuration required. Data retention is 90 days, but you can configure durable storage for longer retention.

1. In the Azure portal, find your search service. From the menu on the left, select **Activity logs** to view the logs for your search service. See Azure Monitor activity log for general guidance on working with activity logs.

2. Review the entries. Entries often include **Get Admin Key**, one entry for every call that provided an admin API key on the request. There are no details about the call itself, just a notification that the admin key was used.

3. For other entires, see the Management REST API reference for control plane activity that might appear in the log.

The following screenshot shows the activity log signals that can be configured in an alert. These signals represent the entries you might see in the activity log.

Monitor metrics

In Azure Cognitive Search, **platform metrics** measure query performance, indexing volume, and skillset invocation.

Metrics are collected free of charge, with no configuration required. Platform metrics are stored for 93 days. However, in the portal you can only query a maximum of 30 days' worth of platform metrics data on any single chart.

In the Azure portal, find your search service. From the menu on the left, under Monitoring, select **Metrics** to open metrics explorer.

The following links provide more information about working with platform metrics:

- Tutorial: Analyze metrics for an Azure resource for general guidance on using metrics explorer.
- Microsoft.Search/searchServices (Azure Monitor) for the platform metrics reference.
- Monitoring data reference for supplemental descriptions and dimensions.
- Monitor queries has details about the query metrics.

Set up alerts

Alerts help you to identify and address issues before they become a problem for application users. You can set alerts on metrics, resource logs, and activity logs. Alerts are billable (see the Pricing model for details).

1. In the Azure portal, find your search service. From the menu on the left, under Monitoring, select **Alerts** to open metrics explorer.
2. See Tutorial: Create a metric alert for an Azure resource for general guidance on setting up alerts from metrics explorer.

The following table describes several rules. On a search service, throttling or query latency that exceeds a given threshold are the most commonly used alerts, but you might also want to be notified if a search service is deleted.

Alert type	Condition	Description
Search Latency (metric alert)	Whenever the average search latency is greater than a user-specified threshold (in seconds)	Send an SMS alert when average query response time exceeds the threshold.
Throttled search queries percentage (metric alert)	Whenever the total throttled search queries percentage is greater than or equal to a user-specified threshold	Send an SMS alert when dropped queries begin to exceed the threshold.
Delete Search Service (activity log alert)	Whenever the Activity Log has an event with Category='Administrative', Signal name='Delete Search Service (searchServices)', Level='critical'	Send an email if a search service is deleted in the subscription.

> **Note**
>
> Currently, there are no storage-related alerts (storage consumption data isn't aggregated or logged into the **AzureMetrics** table). To get storage alerts, you could **build a custom solution** that emits resource-related notifications, where your code checks for storage size and handles the response.

Enable resource logging

In Azure Cognitive Search, **resource logs** capture indexing and query operations on the search service itself.

Resource Logs aren't collected and stored until you create a diagnostic setting. A diagnostic setting specifies data collection and storage. You can create multiple settings if you want to keep metrics and log data separate, or if you want more than one of each type of destination.

Resource logging is billable (see the Pricing model for details), starting when you create a diagnostic setting. See Diagnostic settings in Azure Monitor for general guidance.

1. In the Azure portal, find your search service. From the menu on the left, under Monitoring, select **Diagnostic settings**.
2. Select + **Add diagnostic setting**.
3. Give the diagnostic setting a name. Use granular and descriptive names if you're creating more than one setting.
4. Select the logs and metrics that are in scope for this setting. Selections include "allLogs", "OperationLogs", "AllMetrics". You can exclude activity logs by selecting the "OperationLogs" category.
   - See Microsoft.Search/searchServices (in Supported categories for Azure Monitor resource logs)
   - See Microsoft.Search/searchServices (in Supported metrics)
   - See Azure Cognitive Search monitoring data reference for the extended schema
5. Select **Send to Log Analytics workspace**. Kusto queries and data exploration will target the workspace.

6. Optionally, select other destinations.

7. Select **Save**.

Once the workspace contains data, you can run log queries:

- See Tutorial: Collect and analyze resource logs from an Azure resource for general guidance on log queries.
- See Analyze performance in Azure Cognitive Search for examples and guidance specific to search services.

Sample Kusto queries

> **Important**
>
> When you select **Logs** from the Azure Cognitive Search menu, Log Analytics is opened with the query scope set to the current search service. This means that log queries will only include data from that resource. If you want to query over multiple search services or combine data from other Azure services, select **Logs** from the **Azure Monitor** menu. See **Log query scope and time range in Azure Monitor Log Analytics** for details.

Kusto is the query language used for Log Analytics. The next section has some queries to get you started. See the **Azure Cognitive Search monitoring data reference** for descriptions of schema elements used in a query. See Analyze performance in Azure Cognitive Search for more examples and guidance specific to search service.

List metrics by name

Return a list of metrics and the associated aggregation. The query is scoped to the current search service over the time range that you specify.

Kusto

```
AzureMetrics
| project MetricName, Total, Count, Maximum, Minimum, Average
```

List operations by name

Return a list of operations and a count of each one.

```kusto
AzureDiagnostics
| summarize count() by OperationName
```

Long-running queries

This Kusto query against AzureDiagnostics returns `Query.Search` operations, sorted by duration (in milliseconds). For more examples of `Query.Search` queries, see Analyze performance in Azure Cognitive Search.

```kusto
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s,
Documents_d, IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

Indexer status

This Kusto query returns the status of indexer operations. Results include the operation name, description of the request (which includes the name of the indexer), result status (Success or Failure), and the HTTP status code. For more information about indexer execution, see Monitor indexer status.

```kusto
AzureDiagnostics
| project OperationName, Description_s, Documents_d, ResultType,
resultSignature_d
| where OperationName == "Indexers.Status"
```

Next steps

The monitoring framework for Azure Cognitive Search is provided by Azure Monitor. If you're not familiar with this service, start with Monitoring Azure resources with Azure Monitor to review the main concepts. You can also review the following articles for Azure Cognitive Search:

- Analyze performance in Azure Cognitive Search
- Monitor queries

- Monitor indexer-based indexing

Additional resources

Documentation

Analyze performance - Azure Cognitive Search

TBD

Service limits for tiers and skus - Azure Cognitive Search

Service limits used for capacity planning and maximum limits on requests and responses for Azure Cognitive Search.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Service Statistics in Azure Cognitive Search

Get statistics about the usage of service-level resources in your Azure Cognitive Search service

Monitor indexer status and results - Azure Cognitive Search

Monitor the status, progress, and results of Azure Cognitive Search indexers in the Azure portal, using the REST API, or the .NET SDK.

Reliability in Azure Cognitive Search - Azure Cognitive Search

Find out about reliability in Azure Cognitive Search.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Show 5 more

Training

Learning path

Monitor the usage, performance, and availability of resources with Azure Monitor - Training

Learn how to monitor the usage, performance, and availability of resources using Azure Monitor.

Certification

Microsoft Certified: Azure Administrator Associate - Certifications

Azure administrators implement, manage, and monitor an organization's Microsoft Azure environment, including virtual networks, storage, compute, identity, security, and governance.

Visualize Azure Cognitive Search Logs and Metrics with Power BI

Article • 09/15/2022 • 3 minutes to read

Azure Cognitive Search can send operation logs and service metrics to an Azure Storage account, which you can then visualize in Power BI. This article explains the steps and how to use a Power BI Template App to visualize the data. The template can help you gain detailed insights about your search service, including information about queries, indexing, operations, and service metrics.

You can find the Power BI Template App **Azure Cognitive Search: Analyze Logs and Metrics** in the Power BI Apps marketplace.

Set up the app

1. Enable metric and resource logging for your search service:

   a. Create or identify an existing Azure Storage account where you can archive the logs.

   b. Navigate to your Azure Cognitive Search service in the Azure portal.

   c. Under the Monitoring section on the left column, select **Diagnostic settings**.

   d. Select + **Add diagnostic setting**.

   e. Check **Archive to a storage account**, provide your Storage account information, and check **OperationLogs** and **AllMetrics**.

f. Select **Save**.

2. After logging has been enabled, use your search service to start generating logs and metrics. It takes up to an hour before the containers will appear in Blob storage with these logs. You will see a **insights-logs-operationlogs** container for search traffic logs and a **insights-metrics-pt1m** container for metrics.

3. Find the Azure Cognitive Search Power BI App in the Power BI Apps marketplace and install it into a new workspace or an existing workspace. The app is called **Azure Cognitive Search: Analyze Logs and Metrics**.

4. After installing the app, select the app from your list of apps in Power BI.

5. Select **Connect** to connect your data

6. Input the name of the storage account that contains your logs and metrics. By default the app will look at the last 10 days of data but this value can be changed with the **Days** parameter.

7. Select **Key** as the authentication method and provide your storage account key. Select **Private** as the privacy level. Click Sign In and to begin the loading process.

8. Wait for the data to refresh. This may take some time depending on how much data you have. You can see if the data is still being refreshed based on the below indicator.

9. Once the data refresh has completed, select **Azure Cognitive Search Report** to view the report.

10. Make sure to refresh the page after opening the report so that it opens with your data.

Modify app parameters

If you would like to visualize data from a different storage account or change the number of days of data to query, follow the below steps to change the **Days** and **StorageAccount** parameters.

1. Navigate to your Power BI apps, find your Azure Cognitive Search app and select the **Edit app** button to view the workspace.

2. Select **Settings** from the Dataset options.

3. While in the Datasets tab, change the parameter values and select **Apply**. If there is an issue with the connection, update the data source credentials on the same page.

4. Navigate back to the workspace and select **Refresh now** from the Dataset options.

5. Open the report to view the updated data. You might also need to refresh the report to view the latest data.

Troubleshooting report issues

If you find that you cannot see your data follow these troubleshooting steps:

1. Open the report and refresh the page to make sure you're viewing the latest data. There's an option in the report to refresh the data. Select this to get the latest data.

2. Ensure the storage account name and access key you provided are correct. The storage account name should correspond to the account configured with your search service logs.

3. Confirm that your storage account contains the containers **insights-logs-operationlogs** and **insights-metrics-pt1m** and each container has data. The logs and metrics will be within a couple layers of folders.

4. Check to see if the dataset is still refreshing. The refresh status indicator is shown in step 8 above. If it is still refreshing, wait until the refresh is complete to open and refresh the report.

Next steps

- Monitor search operations and activity
- What is Power BI?
- Basic concepts for designers in the Power BI service

Additional resources

Documentation

Monitor queries - Azure Cognitive Search

Monitor query metrics for performance and throughput. Collect and analyze query string inputs in resource logs.

Azure Cognitive Search monitoring data reference

Log and metrics reference for monitoring data from Azure Cognitive Search.

Monitor Azure Cognitive Search

Enable resource logging, get query metrics, resource usage, and other system data about an Azure Cognitive Search service.

Reliability in Azure Cognitive Search - Azure Cognitive Search

Find out about reliability in Azure Cognitive Search.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Analyze performance - Azure Cognitive Search

TBD

How to move your service resource across regions - Azure Cognitive Search

This article will show you how to move your Azure Cognitive Search resources from one region to another in the Azure cloud.

Show 5 more

Training

Certification

Microsoft Certified: Power BI Data Analyst Associate - Certifications

Power BI data analyst works closely with business stakeholders to identify business requirements. They collaborate with enterprise data analysts and data engineers to identify and acquire data.

Monitor query requests in Azure Cognitive Search

Article • 03/01/2023 • 8 minutes to read

This article explains how to measure query performance and volume using built-in metrics and resource logging. It also explains how to get the query strings entered by application users.

The Azure portal shows basic metrics about query latency, query load (QPS), and throttling. Historical data that feeds into these metrics can be accessed in the portal for 30 days. For longer retention, or to report on operational data and query strings, you must enable a diagnostic setting that specifies a storage option for persisting logged operations and metrics.

Conditions that maximize the integrity of data measurement include:

- Use a billable service (a service created at either the Basic or a Standard tier). The free service is shared by multiple subscribers, which introduces a certain amount of volatility as loads shift.
- Use a single replica and partition, if possible, to create a contained and isolated environment. If you use multiple replicas, query metrics are averaged across multiple nodes, which can lower the precision of results. Similarly, multiple partitions mean that data is divided, with the potential that some partitions might have different data if indexing is also underway. When tuning query performance, a single node and partition gives a more stable environment for testing.

> **Tip**
>
> With additional client-side code and Application Insights, you can also capture clickthrough data for deeper insight into what attracts the interest of your application users. For more information, see **Search traffic analytics**.

Query volume (QPS)

Volume is measured as **Search Queries Per Second** (QPS), a built-in metric that can be reported as an average, count, minimum, or maximum values of queries that execute within a one-minute window. One-minute intervals (TimeGrain = "PT1M") for metrics is fixed within the system.

It's common for queries to execute in milliseconds, so only queries that measure as seconds will appear in metrics.

Aggregation Type	Description
Average	The average number of seconds within a minute during which query execution occurred.
Count	The number of metrics emitted to the log within the one-minute interval.
Maximum	The highest number of search queries per second registered during a minute.
Minimum	The lowest number of search queries per second registered during a minute.
Sum	The sum of all queries executed within the minute.

For example, within one minute, you might have a pattern like this: one second of high load that is the maximum for SearchQueriesPerSecond, followed by 58 seconds of average load, and finally one second with only one query, which is the minimum.

Another example: if a node emits 100 metrics, where the value of each metric is 40, then "Count" is 100, "Sum" is 4000, "Average" is 40, and "Max" is 40.

Query performance

Service-wide, query performance is measured as search latency (how long a query takes to complete) and throttled queries that were dropped as a result of resource contention.

Search latency

Aggregation Type	Latency
Average	Average query duration in milliseconds.
Count	The number of metrics emitted to the log within the one-minute interval.
Maximum	Longest running query in the sample.
Minimum	Shortest running query in the sample.
Total	Total execution time of all queries in the sample, executing within the interval (one minute).

Consider the following example of **Search Latency** metrics: 86 queries were sampled, with an average duration of 23.26 milliseconds. A minimum of 0 indicates some queries were dropped. The longest running query took 1000 milliseconds to complete. Total execution time was 2 seconds.

Throttled queries

Throttled queries refers to queries that are dropped instead of process. In most cases, throttling is a normal part of running the service. It is not necessarily an indication that there is something wrong.

Throttling occurs when the number of requests currently processed exceed the available resources. You might see an increase in throttled requests when a replica is taken out of rotation or during indexing. Both query and indexing requests are handled by the same set of resources.

The service determines whether to drop requests based on resource consumption. The percentage of resources consumed across memory, CPU, and disk IO are averaged over a period of time. If this percentage exceeds a threshold, all requests to the index are throttled until the volume of requests is reduced.

Depending on your client, a throttled request can be indicated in these ways:

- A service returns an error "You are sending too many requests. Please try again later."
- A service returns a 503 error code indicating the service is currently unavailable.
- If you are using the portal (for example, Search Explorer), the query is dropped silently and you will need to click Search again.

To confirm throttled queries, use **Throttled search queries** metric. You can explore metrics in the portal or create an alert metric as described in this article. For queries that were dropped within the sampling interval, use *Total* to get the percentage of queries that did not execute.

Aggregation Type	Throttling
Average	Percentage of queries dropped within the interval.
Count	The number of metrics emitted to the log within the one-minute interval.

Aggregation Type	Throttling
Maximum	Percentage of queries dropped within the interval.
Minimum	Percentage of queries dropped within the interval.
Total	Percentage of queries dropped within the interval.

For **Throttled Search Queries Percentage**, minimum, maximum, average and total, all have the same value: the percentage of search queries that were throttled, from the total number of search queries during one minute.

In the following screenshot, the first number is the count (or number of metrics sent to the log). Additional aggregations, which appear at the top or when hovering over the metric, include average, maximum, and total. In this sample, no requests were dropped.

Explore metrics in the portal

For a quick look at the current numbers, the **Monitoring** tab on the service Overview page shows three metrics (**Search latency**, **Search queries per second (per search unit)**, **Throttled Search Queries Percentage**) over fixed intervals measured in hours, days, and weeks, with the option of changing the aggregation type.

For deeper exploration, open metrics explorer from the **Monitoring** menu so that you can layer, zoom in, and visualize data to explore trends or anomalies. Learn more about metrics explorer by completing this tutorial on creating a metrics chart.

1. Under the Monitoring section, select **Metrics** to open the metrics explorer with the scope set to your search service.

2. Under Metric, choose one from the dropdown list and review the list of available aggregations for a preferred type. The aggregation defines how the collected values will be sampled over each time interval.

3. In the top-right corner, set the time interval.

4. Choose a visualization. The default is a line chart.

5. Layer additional aggregations by choosing **Add metric** and selecting different aggregations.

6. Zoom into an area of interest on the line chart. Put the mouse pointer at the beginning of the area, click and hold the left mouse button, drag to the other side of area, and release the button. The chart will zoom in on that time range.

Return query strings entered by users

When you enable resource logging, the system captures query requests in the **AzureDiagnostics** table. As a prerequisite, you must have already enabled resource logging, specifying a log analytics workspace or another storage option.

1. Under the Monitoring section, select **Logs** to open up an empty query window in Log Analytics.

2. Run the following expression to search Query.Search operations, returning a tabular result set consisting of the operation name, query string, the index queried, and the number of documents found. The last two statements exclude query strings consisting of an empty or unspecified search, over a sample index, which cuts down the noise in your results.

Kusto

```
   AzureDiagnostics
| project OperationName, Query_s, IndexName_s, Documents_d
| where OperationName == "Query.Search"
```

```
| where Query_s != "?api-version=2020-06-30&search=*"
| where IndexName_s != "realestate-us-sample-index"
```

3. Optionally, set a Column filter on *Query_s* to search over a specific syntax or string. For example, you could filter over *is equal to* `?api-version=2020-06-30&search=*&%24filter=HotelName`).

While this technique works for ad hoc investigation, building a report lets you consolidate and present the query strings in a layout more conducive to analysis.

Identify long-running queries

Add the duration column to get the numbers for all queries, not just those that are picked up as a metric. Sorting this data shows you which queries take the longest to complete.

1. Under the Monitoring section, select **Logs** to query for log information.

2. Run the following basic query to return queries, sorted by duration in milliseconds. The longest-running queries are at the top.

Kusto

```
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s,
Documents_d, IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

Create a metric alert

A metric alert establishes a threshold at which you will either receive a notification or trigger a corrective action that you define in advance.

For a search service, it's common to create a metric alert for search latency and throttled queries. If you know when queries are dropped, you can look for remedies that reduce load or increase capacity. For example, if throttled queries increase during indexing, you could postpone it until query activity subsides.

When pushing the limits of a particular replica-partition configuration, setting up alerts for query volume thresholds (QPS) is also helpful.

1. Under the Monitoring section, select **Alerts** and then click **+ New alert rule**. Make sure your search service is selected as the resource.

2. Under Condition, click **Add**.

3. Configure signal logic. For signal type, choose **metrics** and then select the signal.

4. After selecting the signal, you can use a chart to visualize historical data for an informed decision on how to proceed with setting up conditions.

5. Next, scroll down to Alert logic. For proof-of-concept, you could specify an artificially low value for testing purposes.

6. Next, specify or create an Action Group. This is the response to invoke when the threshold is met. It might be a push notification or an automated response.

7. Last, specify Alert details. Name and describe the alert, assign a severity value, and specify whether to create the rule in an enabled or disabled state.

If you specified an email notification, you will receive an email from "Microsoft Azure" with a subject line of "Azure: Activated Severity: 3 `<your rule name>`".

Next steps

If you haven't done so already, review the fundamentals of search service monitoring to learn about the full range of oversight capabilities.

Monitor operations and activity in Azure Cognitive Search

Additional resources

Documentation

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Monitor indexer status and results - Azure Cognitive Search

Monitor the status, progress, and results of Azure Cognitive Search indexers in the Azure portal, using the REST API, or the .NET SDK.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Index large data set using built-in indexers - Azure Cognitive Search

Strategies for large data indexing or computationally intensive indexing through batch mode, resourcing, and techniques for scheduled, parallel, and distributed indexing.

Analyze performance - Azure Cognitive Search

TBD

Monitor Azure Cognitive Search

Enable resource logging, get query metrics, resource usage, and other system data about an Azure Cognitive Search service.

Indexer troubleshooting guidance - Azure Cognitive Search

This article provides indexer problem and resolution guidance for cases when no error messages are returned from the service search.

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Show 5 more

Monitor indexer status and results in Azure Cognitive Search

Article • 09/15/2022 • 5 minutes to read

You can monitor indexer processing in the Azure portal, or programmatically through REST calls or an Azure SDK. In addition to status about the indexer itself, you can review start and end times, and detailed errors and warnings from a particular run.

Monitor using Azure portal

You can see the current status of all of your indexers in your search service Overview page. Portal pages refresh every few minutes, so you won't see evidence of a new indexer run right away. Select **Refresh** at the top of the page to immediately retrieve the most recent view.

Status	Description
In Progress	Indicates active execution. The portal will report on partial information. As indexing progresses, you can watch the **Docs Succeeded** value grow in response. Indexers that process large volumes of data can take a long time to run. For example, indexers that handle millions of source documents can run for 24 hours, and then restart almost immediately to pick up where it left off. As such, the status for high-volume indexers might always say **In Progress** in the portal. Even when an indexer is running, details are available about ongoing progress and previous runs.
Success	Indicates the run was successful. An indexer run can be successful even if individual documents have errors, if the number of errors is less than the indexer's **Max failed items** setting.
Failed	The number of errors exceeded **Max failed items** and indexing has stopped.
Reset	The indexer's internal change tracking state was reset. The indexer will run in full, refreshing all documents, and not just those with newer timestamps.

You can select on an indexer in the list to see more details about the indexer's current and recent runs.

The **Indexer summary** chart displays a graph of the number of documents processed in its most recent runs.

The **Execution details** list shows up to 50 of the most recent execution results. Select on an execution result in the list to see specifics about that run. This includes its start and end times, and any errors and warnings that occurred.

If there were document-specific problems during the run, they'll be listed in the Errors and Warnings fields.

Warnings are common with some types of indexers, and don't always indicate a problem. For example indexers that use Cognitive Services can report warnings when image or PDF files don't contain any text to process.

For more information about investigating indexer errors and warnings, see Indexer troubleshooting guidance.

Monitor with Azure Monitoring Metrics

Cognitive Search is a monitored resource in Azure Monitor, which means that you can use Metrics Explorer to see basic metrics about the number of indexer-processed

documents and skill invocations. These metrics can be used to monitor indexer progress and set up alerts.

Metric views can be filtered or split up by a set of predefined dimensions.

Metric Name	Description	Dimensions	Sample use cases
Document processed count	Shows the number of indexer processed documents.	Data source name, failed, index name, indexer name, skillset name	- Can be referenced as a rough measure of throughput (number of documents processed by indexer over time) - Set up to alert on failed documents
Skill execution invocation count	Shows the number of skill invocations.	Data source name, failed, index name, indexer name, skill name, skill type, skillset name	- Reference to ensure skills are invoked as expected by comparing relative invocation numbers between skills and number of skill invocations to the number of documents. - Set up to alert on failed skill invocations

The screenshot below shows the number of documents processed by indexers within a service over an hour, split up by indexer name.

You can also configure the graph to see the number of skill invocations over the same hour interval.

Monitor using Get Indexer Status (REST API)

You can retrieve the status and execution history of an indexer using the Get Indexer Status command:

HTTP

```
GET https://[service name].search.windows.net/indexers/[indexer name]/status?api-version=2020-06-30
api-key: [Search service admin key]
```

The response contains overall indexer status, the last (or in-progress) indexer invocation, and the history of recent indexer invocations.

Output

```
{
    "status":"running",
    "lastResult": {
        "status":"success",
        "errorMessage":null,
        "startTime":"2018-11-26T03:37:18.853Z",
        "endTime":"2018-11-26T03:37:19.012Z",
        "errors":[],
        "itemsProcessed":11,
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
     },
    "executionHistory":[ {
        "status":"success",
         "errorMessage":null,
        "startTime":"2018-11-26T03:37:18.853Z",
        "endTime":"2018-11-26T03:37:19.012Z",
        "errors":[],
```

```
        "itemsProcessed":11,
        "itemsFailed":0,
        "initialTrackingState":null,
        "finalTrackingState":null
    }]
}
```

Execution history contains up to the 50 most recent runs, which are sorted in reverse chronological order (most recent first).

Note there are two different status values. The top level status is for the indexer itself. An indexer status of **running** means the indexer is set up correctly and available to run, but not that it's currently running.

Each run of the indexer also has its own status that indicates whether that specific execution is ongoing (**running**), or already completed with a **success**, **transientFailure**, or **persistentFailure** status.

When an indexer is reset to refresh its change tracking state, a separate execution history entry is added with a **Reset** status.

For more information about status codes and indexer monitoring data, see Get Indexer Status.

Monitor using .NET

Using the Azure Cognitive Search .NET SDK, the following C# example writes information about an indexer's status and the results of its most recent (or ongoing) run to the console.

C#

```
static void CheckIndexerStatus(SearchIndexerClient indexerClient,
SearchIndexer indexer)
{
    try
    {
        string indexerName = "hotels-sql-idxr";
        SearchIndexerStatus execInfo =
indexerClient.GetIndexerStatus(indexerName);

        Console.WriteLine("Indexer has run {0} times.",
execInfo.ExecutionHistory.Count);
        Console.WriteLine("Indexer Status: " + execInfo.Status.ToString());

        IndexerExecutionResult result = execInfo.LastResult;
```

```
        Console.WriteLine("Latest run");
        Console.WriteLine("Run Status: {0}", result.Status.ToString());
        Console.WriteLine("Total Documents: {0}, Failed: {1}",
result.ItemCount, result.FailedItemCount);

        TimeSpan elapsed = result.EndTime.Value - result.StartTime.Value;
        Console.WriteLine("StartTime: {0:T}, EndTime: {1:T}, Elapsed:
{2:t}", result.StartTime.Value, result.EndTime.Value, elapsed);

        string errorMsg = (result.ErrorMessage == null) ? "none" :
result.ErrorMessage;
        Console.WriteLine("ErrorMessage: {0}", errorMsg);
        Console.WriteLine(" Document Errors: {0}, Warnings: {1}\n",
result.Errors.Count, result.Warnings.Count);
    }
    catch (Exception e)
    {
        // Handle exception
    }
}
```

The output in the console will look something like this:

Output

```
Indexer has run 18 times.
Indexer Status: Running
Latest run
  Run Status: Success
  Total Documents: 7, Failed: 0
  StartTime: 11:29:31 PM, EndTime: 11:29:31 PM, Elapsed: 00:00:00.2560000
  ErrorMessage: none
  Document Errors: 0, Warnings: 0
```

Note there are two different status values. The top-level status is the status of the indexer itself. An indexer status of **Running** means that the indexer is set up correctly and available for execution, but not that it's currently executing.

Each run of the indexer also has its own status for whether that specific execution is ongoing (**Running**), or was already completed with a **Success** or **TransientError** status.

When an indexer is reset to refresh its change tracking state, a separate history entry is added with a **Reset** status.

Next steps

For more information about status codes and indexer monitoring information, see the following API reference:

- GetIndexerStatus (REST API)
- IndexerStatus
- IndexerExecutionStatus
- IndexerExecutionResult

Additional resources

Documentation

Reset Indexer (Azure Cognitive Search REST API)

Rebuild an index created by an Azure Cognitive Search indexer.

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Index operations using Azure Cognitive Search REST APIs

Learn which REST APIs are used to create, delete, and update Azure Cognitive Search index definitions.

Create an index alias - Azure Cognitive Search

Create an alias to define a secondary name that can be used to refer to an index for querying, indexing, and other operations.

Analyze performance - Azure Cognitive Search

TBD

Run Indexer (Azure Cognitive Search REST API)

Trigger Azure Cognitive Search indexing when using an indexer to crawl external data for searchable content.

Add scoring profiles - Azure Cognitive Search

Boost search relevance scores for Azure Cognitive Search results by adding scoring profiles to a search index.

Create Index (Azure Cognitive Search REST API)

Define an index schema for fields and other constructs in an Azure Cognitive Search index.

Show 5 more

Collect telemetry data for search traffic analytics

Article • 05/26/2023

Search traffic analytics is a pattern for collecting telemetry about user interactions with your Azure Cognitive Search application, such as user-initiated click events and keyboard inputs. Using this information, you can determine the effectiveness of your search solution, including popular search terms, clickthrough rate, and which query inputs yield zero results.

This pattern takes a dependency on Application Insights (a feature of Azure Monitor) to collect user data. It requires that you add instrumentation to your client code, as described in this article. Finally, you will need a reporting mechanism to analyze the data. We recommend Power BI, but you can use the Application Dashboard or any tool that connects to Application Insights.

> **Note**
>
> The pattern described in this article is for advanced scenarios and clickstream data generated by code you add to your client. In contrast, service logs are easy to set up, provide a range of metrics, and can be done in the portal with no code required. Enabling logging is recommended for all scenarios. For more information, see **Collect and analyze log data**.

Identify relevant search data

To have useful metrics for search traffic analytics, it's necessary to log some signals from the users of your search application. These signals signify content that users are interested in and that they consider relevant. For search traffic analytics, these include:

- User-generated search events: Only search queries initiated by a user are interesting. Other search requests, such as those used to populate facets or retrieve internal information, are not important. Be sure to only instrument user-initiated events to avoid skew or bias in your results.
- User-generated click events: On a search results page, a click event generally means that a document is a relevant result for a specific search query.

By linking search and click events with a correlation ID, you'll gain a deeper understanding of how well your application's search functionality is performing.

Add search traffic analytics

In the portal page for your Azure Cognitive Search service, open the Search Traffic Analytics page to access a cheat sheet for following this telemetry pattern. From this page, you can select or create an Application Insights resource, get the instrumentation key, copy snippets that you can adapt for your solution, and download a Power BI report that's built over the schema reflected in the pattern.

1 - Set up Application Insights

Select an existing Application Insights resource or create one if you don't have one already. If you use the Search Traffic Analytics page, you can copy the instrumentation key your application needs to connect to Application Insights.

Once you have an Application Insights resource, you can follow instructions for supported languages and platforms to register your app. Registration is simply adding the instrumentation key from Application Insights to your code, which sets up the association. You can find the key in the portal, or from the Search Traffic Analytics page when you select an existing resource.

A shortcut that works for some Visual Studio project types is reflected in the following steps. It creates a resource and registers your app in just a few clicks.

1. For Visual Studio and ASP.NET development, open your solution and select **Project** > **Add Application Insights Telemetry**.
2. Click **Get Started**.
3. Register your app by providing a Microsoft account, Azure subscription, and an Application Insights resource (a new resource is the default). Click **Register**.

At this point, your application is set up for application monitoring, which means all page loads are tracked with default metrics. For more information about the previous steps, see Enable Application Insights server-side telemetry.

2 - Add instrumentation

This step is where you instrument your own search application, using the Application Insights resource your created in the step above. There are four steps to this process, starting with creating a telemetry client.

Step 1: Create a telemetry client

Create an object that sends events to Application Insights. You can add instrumentation to your server-side application code or client-side code running in a browser, expressed here as C# and JavaScript variants (for other languages, see the complete list of supported platforms and frameworks. Choose the approach that gives you the desired depth of information.

Server-side telemetry captures metrics at the application layer, for example in applications running as a web service in the cloud, or as an on-premises app on a corporate network. Server-side telemetry captures search and click events, the position of a document in results, and query information, but your data collection will be scoped to whatever information is available at that layer.

On the client, you might have additional code that manipulates query inputs, adds navigation, or includes context (for example, queries initiated from a home page versus a product page). If this describes your solution, you might opt for client-side instrumentation so that your telemetry reflects the additional detail. How this additional detail is collected goes beyond the scope of this pattern, but you can review Application Insights for web pages for more direction.

Use C#

For C#, the **InstrumentationKey** should be defined in your application configuration, such as appsettings.json if your project is ASP.NET. Refer back to the registration instructions if you are unsure of the key location.

C#

```
private static TelemetryClient _telemetryClient;

// Add a constructor that accepts a telemetry client:
public HomeController(TelemetryClient telemetry)
{
    _telemetryClient = telemetry;
}
```

Use JavaScript

To create an object that sends events to Application Insights by using the SDK Loader Script, see Microsoft Azure Monitor Application Insights JavaScript SDK.

Step 2: Request a Search ID for correlation

To correlate search requests with clicks, it's necessary to have a correlation ID that relates these two distinct events. Azure Cognitive Search provides you with a search ID when you request it with an HTTP header.

Having the search ID allows correlation of the metrics emitted by Azure Cognitive Search for the request itself, with the custom metrics you are logging in Application Insights.

Use C# (newer v11 SDK)

The latest SDK requires the use of an Http Pipeline to set the header as detailed in this sample.

C#

```
// Create a custom policy to add the correct headers
public class SearchIdPipelinePolicy : HttpPipelineSynchronousPolicy
{
    public override void OnSendingRequest(HttpMessage message)
    {
        message.Request.Headers.SetValue("x-ms-azs-return-searchid",
"true");
    }
}
```

C#

```
// This sample uses the .NET SDK
https://www.nuget.org/packages/Azure.Search.Documents

SearchClientOptions clientOptions = new SearchClientOptions();
clientOptions.AddPolicy(new SearchIdPipelinePolicy(),
HttpPipelinePosition.PerCall);

var client = new SearchClient("<SearchServiceName>", "<IndexName>", new
AzureKeyCredential("<QueryKey>"), options: clientOptions);

Response<SearchResults<SearchDocument>> response = await
client.SearchAsync<SearchDocument>(searchText: searchText, searchOptions:
options);
string searchId = string.Empty;
if (response.GetRawResponse().Headers.TryGetValues("x-ms-azs-searchid", out
IEnumerable<string> headerValues))
{
    searchId = headerValues.FirstOrDefault();
}
```

Use C# (older v10 SDK)

C#

```
// This sample uses the .NET SDK
https://www.nuget.org/packages/Microsoft.Azure.Search

var client = new SearchIndexClient(<SearchServiceName>, <IndexName>, new
SearchCredentials(<QueryKey>));

// Use HTTP headers so that you can get the search ID from the response
var headers = new Dictionary<string, List<string>>() { { "x-ms-azs-return-
searchid", new List<string>() { "true" } } };
var response = await
client.Documents.SearchWithHttpMessagesAsync(searchText: searchText,
searchParameters: parameters, customHeaders: headers);
string searchId = string.Empty;
if (response.Response.Headers.TryGetValues("x-ms-azs-searchid", out
IEnumerable<string> headerValues))
{
    searchId = headerValues.FirstOrDefault();
}
```

Use JavaScript (calling REST APIs)

JavaScript

```
request.setRequestHeader("x-ms-azs-return-searchid", "true");
request.setRequestHeader("Access-Control-Expose-Headers", "x-ms-azs-
```

```
searchid");
var searchId = request.getResponseHeader('x-ms-azs-searchid');
```

Step 3: Log Search events

Every time that a search request is issued by a user, you should log that as a search event with the following schema on an Application Insights custom event. Remember to log only user-generated search queries.

- **SearchServiceName**: (string) search service name
- **SearchId**: (guid) unique identifier of the search query (comes in the search response)
- **IndexName**: (string) search service index to be queried
- **QueryTerms**: (string) search terms entered by the user
- **ResultCount**: (int) number of documents that were returned (comes in the search response)
- **ScoringProfile**: (string) name of the scoring profile used, if any

> **Note**
>
> Request the count of user generated queries by adding $count=true to your search query. For more information, see Search Documents (REST).

Use C#

C#

```
var properties = new Dictionary <string, string>
{
    {"SearchServiceName", <service name>},
    {"SearchId", <search Id>},
    {"IndexName", <index name>},
    {"QueryTerms", <search terms>},
    {"ResultCount", <results count>},
    {"ScoringProfile", <scoring profile used>}
};
_telemetryClient.TrackEvent("Search", properties);
```

Use JavaScript

JavaScript

```
appInsights.trackEvent("Search", {
  SearchServiceName: <service name>,
```

```
  SearchId: <search id>,
  IndexName: <index name>,
  QueryTerms: <search terms>,
  ResultCount: <results count>,
  ScoringProfile: <scoring profile used>
});
```

Step 4: Log Click events

Every time that a user clicks on a document, that's a signal that must be logged for search analysis purposes. Use Application Insights custom events to log these events with the following schema:

- **ServiceName**: (string) search service name
- **SearchId**: (guid) unique identifier of the related search query
- **DocId**: (string) document identifier
- **Position**: (int) rank of the document in the search results page

> **Note**
>
> Position refers to the cardinal order in your application. You are free to set this number, as long as it's always the same, to allow for comparison.

Use C#

C#

```
var properties = new Dictionary <string, string>
{
    {"SearchServiceName", <service name>},
    {"SearchId", <search id>},
    {"ClickedDocId", <clicked document id>},
    {"Rank", <clicked document position>}
};
_telemetryClient.TrackEvent("Click", properties);
```

Use JavaScript

JavaScript

```
appInsights.trackEvent("Click", {
    SearchServiceName: <service name>,
    SearchId: <search id>,
    ClickedDocId: <clicked document id>,
    Rank: <clicked document position>
});
```

3 - Analyze in Power BI

After you have instrumented your app and verified your application is correctly connected to Application Insights, you download a predefined report template to analyze data in Power BI desktop. The report contains predefined charts and tables useful for analyzing the additional data captured for search traffic analytics.

1. In the Azure Cognitive Search dashboard left-navigation pane, under **Settings**, click **Search traffic analytics**.

2. On the **Search traffic analytics** page, in step 3, click **Get Power BI Desktop** to install Power BI.

3. On the same page, click **Download Power BI report**.

4. The report opens in Power BI Desktop, and you are prompted to connect to Application Insights and provide credentials. You can find connection information in the Azure portal pages for your Application Insights resource. For credentials, provide the same user name and password that you use for portal sign-in.

5. Click **Load**.

The report contains charts and tables that help you make more informed decisions to improve your search performance and relevance.

Metrics included the following items:

- Search volume and most popular term-document pairs: terms that result in the same document clicked, ordered by clicks.
- Searches without clicks: terms for top queries that register no clicks

The following screenshot shows what a built-in report might look like if you have used all of the schema elements.

Next steps

Instrument your search application to get powerful and insightful data about your search service.

You can find more information on Application Insights and visit the pricing page to learn more about their different service tiers.

Learn more about creating amazing reports. See Getting started with Power BI Desktop for details.

Azure Cognitive Search Service REST API reference

Article • 05/09/2023

Azure Cognitive Search is a fully managed cloud search service that provides a rich search experience over user-owned content. One way to add search capability is through the Search REST APIs, with operations that create and manage indexes, implement search features, and execute queries.

A separate Management REST API is used to create or modify the service itself. Alternatively, you can use the portal for many service and content management tasks. To get started, see Create a search service in Azure portal.

Key concepts

Cognitive Search has the concepts of *search services*, *indexes*, *documents*, *indexers*, *data sources*, *skillsets*, and *synonym maps*.

- A search service hosts one or more indexes. It also hosts indexers, data sources, skillsets, and synonym maps as top-level objects.
- A search index provides persistent storage of search documents. Search documents are your data, articulated as a collection of fields, loaded from external sources in the form of JSON documents and pushed to an index to make it searchable.
- A search indexer adds automation, reading data in native formats and serializing it into JSON. An indexer has a data source and points to an index. Optionally, it might also have a skillset that adds AI enrichment to the indexer pipeline. Skillsets are always attached to an indexer. They add machine learning transformations that extract or generate text, or add structure to content so that it can be indexed by a search service.

Altogether, there are five types of operations that can be executed against the service:

Operation	Description
Index	Create, delete, update, or configure a search index.
Document	Add, update, or delete documents in the index, query the index, or look up specific documents by ID.

Operation	Description
Indexer	Automate aspects of an indexing operation by configuring a **data source** and an **indexer** that you can schedule or run on demand. This feature is supported for a limited number of data source types on Azure.
Skillset	Part of an AI enrichment workload, a skillset defines a series of enrichment processing that extracts or creates searchable text from unstructured text, application files, or image files. A skillset is invoked by an indexer.
Synonym map	A synonym map is service-level resource that contains user-defined synonyms. This resource is maintained independently from search indexes. Once uploaded, you can point any searchable field to the synonym map (one per field).

Permissions and access control

You can use key-based authentication or role-based through Azure Active Directory (Azure AD).

- Key-based authentication relies on API keys that are generated for the search service. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it. You can use an **Admin API key** for read-write operations or a **Query API key** for read access to the documents collection of a search index.

- Azure AD authentication and role-based access control requires that you have an established tenant in Azure Active Directory, with security principals and role assignments. Members of the following roles have data plane access. You can create custom roles if the built-in roles are insufficient.

Role	Access
Search Service Contributor	Access to objects, but no access to index content. This role can't query a search index or add, remove, or update documents in a search index. This role is for administrators who need to manage objects, but without the ability to view or access object data.
Search Data Index Contributor	Read-write access to index content. This role is for developers or index owners who need to import, refresh, or query the documents collection of an index.
Search Data Index Reader	Read access to index content. This role is for apps and users who run queries.

When using Azure AD on the connection, your client app presents a bearer token in the authorization header. See Authorize access to a search app using Azure Active Directory for help with setting this up.

You can disable key-based authentication or Azure RBAC. If you disable RBAC, some built-in roles will still have access. See Azure AD authentication and role-based access control for Azure Cognitive Search for details.

Calling the APIs

The APIs documented in this section provide access to operations on search data, such as index creation and population, document upload, and queries. When calling APIs, keep the following points in mind:

- Requests must be issued over HTTPS (on the default port 443).
- Requests must include the **api-version** in the URI. The value must be set to a supported version, formatted as shown in this example: `GET https://[search service name].search.windows.net/indexes?api-version=2020-06-30`
- Request headers must include either an **api-key** or a bearer token for authenticated connections. Optionally, you can set the Accept HTTP header. If the header isn't set, the default is assumed to be `application/json`.

See also

- Create a search service in Azure portal
- Common HTTP request and response headers

Search Management

Article • 05/09/2023

The Management REST APIs of Azure Cognitive Search provide programmatic access to administrative operations:

- Create, update, and delete a search service
- Retrieve search service information
- Create, regenerate, or retrieve `api-keys` (query or admin keys)
- Add or remove replicas and partitions (adjust capacity)
- Configure a search service to use a private endpoint
- Enable or disable Azure role-based access control

For all other tasks, such as creating and querying an index, use the Search Service REST APIs instead.

To fully administer your service programmatically, you work with two APIs: the Management REST API of Azure Cognitive Search documented here, plus the common Azure Resource Manager REST API. The Resource Manager API is used for general-purpose operations that aren't service specific, such as querying subscription data, listing geo-locations, and so forth.

Permissions and access control

Administration rights are conveyed through built-in roles in Azure Active Directory (Azure AD):

- **Owner** or **Contributor** assignments are required for most tasks.
- **Reader** role has limited access to service information (GET operations, except for API keys which require Owner or Contributor permissions).

For more information, see Use Azure role-based access controls in Azure Cognitive Search.

A best practice is to assign roles to groups rather than individual users. An Azure subscription owner or global administrator manages these assignments. For more information, see Assign Azure AD roles to groups.

Connect to the management endpoint

When setting up the connection, start with the Resource Manager endpoint `https://management.azure.com` and then add the subscription ID, provider (`Microsoft.Search`), and API version. Operations are specified in the body of the request.

A fully specified endpoint has the following components:

```http
https://management.azure.com/subscriptions/[subscriptionId]/resourceGroups/[resourceGroupName]/providers/Microsoft.Search/searchServices/[serviceName]?api-version=2020-08-01
```

The following clients are commonly used to call the management REST APIs:

Client	Instructions and examples
Azure PowerShell using the Az.Search module	Manage Cognitive Search using Azure PowerShell
Azure Command-Line Interface (CLI)	Manage Cognitive Search using the Azure CLI
Postman	Manage Cognitive Search using REST

You can also manage a search service using the Azure portal.

Management APIs in other SDKs

The REST APIs are foundational to Azure Cognitive Search, and any changes to the programming model are reflected in the REST APIs first.

Alternatives include the following Azure SDKs, which are independently updated on separate release schedules:

- .NET
- Python
- Node.js (readme)
- TypeScript (readme)

See also

- Azure Cognitive Search documentation
- Manage Azure Cognitive Search with REST
- Manage Azure Cognitive Search with Azure PowerShell
- Manage Azure Cognitive Search with Azure CLI

- Manage Azure Cognitive Search using Azure portal
- Scale an Azure Cognitive Search service

Azure Cognitive Search client library for .NET - version 11.4.0

Article • 10/11/2022

Azure Cognitive Search is a search-as-a-service cloud solution that gives developers APIs and tools for adding a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

The Azure Cognitive Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and large text documents. A skillset leverages AI from Cognitive Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines.

Use the Azure.Search.Documents client library to:

- Submit queries for simple and advanced query forms that include fuzzy search, wildcard search, regular expressions.
- Implement filtered queries for faceted navigation, geospatial search, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through scoring profiles to factor in business logic or freshness.

Source code | Package (NuGet) | API reference documentation | REST API documentation | Product documentation | Samples

Getting started

Install the package

Install the Azure Cognitive Search client library for .NET with NuGet:

.NET CLI

```
dotnet add package Azure.Search.Documents
```

Prerequisites

You need an Azure subscription and a search service to use this package.

To create a new search service, you can use the Azure portal, Azure PowerShell, or the Azure CLI. Here's an example using the Azure CLI to create a free instance for getting started:

Powershell

```
az search service create --name <mysearch> --resource-group <mysearch-rg> --
sku free --location westus
```

See choosing a pricing tier for more information about available options.

Authenticate the client

All requests to a search service need an api-key that was generated specifically for your service. The api-key is the sole mechanism for authenticating access to your search service endpoint. You can obtain your api-key from the Azure portal or via the Azure CLI:

Powershell

```
az search admin-key show --service-name <mysearch> --resource-group
<mysearch-rg>
```

There are two types of keys used to access your search service: **admin** *(read-write)* and **query** *(read-only)* keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

We can use the api-key to create a new `SearchClient`.

C#

```
string indexName = "nycjobs";

// Get the service endpoint and API key from the environment
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");

// Create a client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchClient client = new SearchClient(endpoint, indexName, credential);
```

ASP.NET Core

To inject `SearchClient` as a dependency in an ASP.NET Core app, first install the package `Microsoft.Extensions.Azure`. Then register the client in the `Startup.ConfigureServices` method:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAzureClients(builder =>
    {
        builder.AddSearchClient(Configuration.GetSection("SearchClient"));
    });

    services.AddControllers();
}
```

To use the preceding code, add this to your configuration:

JSON

```
{
    "SearchClient": {
      "endpoint": "https://<resource-name>.search.windows.net",
      "indexname": "nycjobs"
    }
}
```

You'll also need to provide your resource key to authenticate the client, but you shouldn't be putting that information in the configuration. Instead, when in development, use User-Secrets. Add the following to `secrets.json`:

JSON

```
{
    "SearchClient": {
      "credential": { "key": "<you resource key>" }
    }
}
```

When running in production, it's preferable to use environment variables:

```

SEARCH__CREDENTIAL__KEY="..."
```

Or use other secure ways of storing secrets like Azure Key Vault.

For more details about Dependency Injection in ASP.NET Core apps, see Dependency injection with the Azure SDK for .NET.

Key concepts

An Azure Cognitive Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. *(If you're brand new to search, you can make a very rough analogy between indexes and database tables.)* The Azure.Search.Documents client library exposes operations on these resources through three main client types.

- `SearchClient` helps with:
  - Searching your indexed documents using rich queries and powerful data shaping
  - Autocompleting partially typed search terms based on documents in the index
  - Suggesting the most likely matching text in documents as a user types
  - Adding, Updating or Deleting Documents documents from an index

- `SearchIndexClient` allows you to:
  - Create, delete, update, or configure a search index
  - Declare custom synonym maps to expand or rewrite queries

- `SearchIndexerClient` allows you to:
  - Create indexers to automatically crawl data sources
  - Define AI powered Skillsets to transform and enrich your data

The `Azure.Search.Documents` client library (v11) is a brand new offering for .NET developers who want to use search technology in their applications. There is an older, fully

featured `Microsoft.Azure.Search` client library (v10) with many similar looking APIs, so please be careful to avoid confusion when exploring online resources. A good rule of thumb is to check for the namespace `using Azure.Search.Documents;` when you're looking for us.

Thread safety

We guarantee that all client instance methods are thread-safe and independent of each other (guideline). This ensures that the recommendation of reusing client instances is always safe, even across threads.

Additional concepts

Client options | Accessing the response | Long-running operations | Handling failures | Diagnostics | Mocking | Client lifetime

Examples

The following examples all use a simple Hotel data set that you can import into your own index from the Azure portal. These are just a few of the basics - please check out our Samples for much more.

- Querying
  - Use C# types for search results
  - Use SearchDocument like a dictionary for search results
  - SearchOptions
- Creating an index
- Adding documents to your index
- Retrieving a specific document from your index
- Async APIs

Advanced authentication

- Create a client that can authenticate in a national cloud

Querying

Let's start by importing our namespaces.

```
C#
```

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
```

We'll then create a `SearchClient` to access our hotels search index.

```C#
// Get the service endpoint and API key from the environment
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");
string indexName = "hotels";

// Create a client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchClient client = new SearchClient(endpoint, indexName, credential);
```

There are two ways to interact with the data returned from a search query. Let's explore them with a search for a "luxury" hotel.

Use C# types for search results

We can decorate our own C# types with attributes from System.Text.Json:

```C#
public class Hotel
{
    [JsonPropertyName("HotelId")]
    [SimpleField(IsKey = true, IsFilterable = true, IsSortable = true)]
    public string Id { get; set; }

    [JsonPropertyName("HotelName")]
    [SearchableField(IsFilterable = true, IsSortable = true)]
    public string Name { get; set; }
}
```

Then we use them as the type parameter when querying to return strongly-typed search results:

```C#
SearchResults<Hotel> response = client.Search<Hotel>("luxury");
foreach (SearchResult<Hotel> result in response.GetResults())
{
    Hotel doc = result.Document;
```

```
    Console.WriteLine($"{doc.Id}: {doc.Name}");
}
```

If you're working with a search index and know the schema, creating C# types is recommended.

Use `SearchDocument` like a dictionary for search results

If you don't have your own type for search results, `SearchDocument` can be used instead. Here we perform the search, enumerate over the results, and extract data using `SearchDocument`'s dictionary indexer.

```C#
SearchResults<SearchDocument> response = client.Search<SearchDocument>
("luxury");
foreach (SearchResult<SearchDocument> result in response.GetResults())
{
    SearchDocument doc = result.Document;
    string id = (string)doc["HotelId"];
    string name = (string)doc["HotelName"];
    Console.WriteLine("{id}: {name}");
}
```

SearchOptions

The `SearchOptions` provide powerful control over the behavior of our queries. Let's search for the top 5 luxury hotels with a good rating.

```C#
int stars = 4;
SearchOptions options = new SearchOptions
{
    // Filter to only Rating greater than or equal our preference
    Filter = SearchFilter.Create($"Rating ge {stars}"),
    Size = 5, // Take only 5 results
    OrderBy = { "Rating desc" } // Sort by Rating from high to low
};
SearchResults<Hotel> response = client.Search<Hotel>("luxury", options);
// ...
```

Creating an index

You can use the `SearchIndexClient` to create a search index. Fields can be defined from a model class using `FieldBuilder`. Indexes can also define suggesters, lexical analyzers, and more:

C#

```
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
string key = Environment.GetEnvironmentVariable("SEARCH_API_KEY");

// Create a service client
AzureKeyCredential credential = new AzureKeyCredential(key);
SearchIndexClient client = new SearchIndexClient(endpoint, credential);

// Create the index using FieldBuilder.
SearchIndex index = new SearchIndex("hotels")
{
    Fields = new FieldBuilder().Build(typeof(Hotel)),
    Suggesters =
    {
        // Suggest query terms from the hotelName field.
        new SearchSuggester("sg", "hotelName")
    }
};

client.CreateIndex(index);
```

In scenarios when the model is not known or cannot be modified, you can also create fields explicitly using convenient `SimpleField`, `SearchableField`, or `ComplexField` classes:

C#

```
// Create the index using field definitions.
SearchIndex index = new SearchIndex("hotels")
{
    Fields =
    {
        new SimpleField("hotelId", SearchFieldDataType.String) { IsKey =
true, IsFilterable = true, IsSortable = true },
        new SearchableField("hotelName") { IsFilterable = true, IsSortable =
true },
        new SearchableField("description") { AnalyzerName =
LexicalAnalyzerName.EnLucene },
        new SearchableField("tags", collection: true) { IsFilterable = true,
IsFacetable = true },
        new ComplexField("address")
        {
            Fields =
            {
                new SearchableField("streetAddress"),
                new SearchableField("city") { IsFilterable = true,
```

```
IsSortable = true, IsFacetable = true },
                new SearchableField("stateProvince") { IsFilterable = true,
IsSortable = true, IsFacetable = true },
                new SearchableField("country") { IsFilterable = true,
IsSortable = true, IsFacetable = true },
                new SearchableField("postalCode") { IsFilterable = true,
IsSortable = true, IsFacetable = true }
            }
        }
    },
    Suggesters =
    {
        // Suggest query terms from the hotelName field.
        new SearchSuggester("sg", "hotelName")
    }
};

client.CreateIndex(index);
```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are a few special rules for merging to be aware of.

C#

```
IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
    IndexDocumentsAction.Upload(new Hotel { Id = "783", Name = "Upload Inn"
}),
    IndexDocumentsAction.Merge(new Hotel { Id = "12", Name = "Renovated
Ranch" }));

IndexDocumentsOptions options = new IndexDocumentsOptions { ThrowOnAnyError
= true };
client.IndexDocuments(batch, options);
```

The request will succeed even if any of the individual actions fail and return an `IndexDocumentsResult` for inspection. There's also a `ThrowOnAnyError` option if you only care about success or failure of the whole batch.

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

```C#
Hotel doc = client.GetDocument<Hotel>("1");
Console.WriteLine($"{doc.Id}: {doc.Name}");
```

Async APIs

All of the examples so far have been using synchronous APIs, but we provide full support for async APIs as well. You'll generally just add an `Async` suffix to the name of the method and `await` it.

```C#
SearchResults<Hotel> response = await client.SearchAsync<Hotel>("luxury");
await foreach (SearchResult<Hotel> result in response.GetResultsAsync())
{
    Hotel doc = result.Document;
    Console.WriteLine($"{doc.Id}: {doc.Name}");
}
```

Authenticate in a National Cloud

To authenticate in a National Cloud, you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `Audience` in `SearchClientOptions`

```C#
// Create a SearchClient that will authenticate through AAD in the China
national cloud
string indexName = "nycjobs";
Uri endpoint = new
Uri(Environment.GetEnvironmentVariable("SEARCH_ENDPOINT"));
SearchClient client = new SearchClient(endpoint, indexName,
    new DefaultAzureCredential(
        new DefaultAzureCredentialOptions()
        {
            AuthorityHost = AzureAuthorityHosts.AzureChina
        }),
    new SearchClientOptions()
    {
        Audience = SearchAudience.AzureChina
    });
```

Troubleshooting

Any Azure.Search.Documents operation that fails will throw a RequestFailedException with helpful Status codes. Many of these errors are recoverable.

```C#
try
{
    return client.GetDocument<Hotel>("12345");
}
catch (RequestFailedException ex) when (ex.Status == 404)
{
    Console.WriteLine("We couldn't find the hotel you are looking for!");
    Console.WriteLine("Please try selecting another.");
    return null;
}
```

You can also easily enable console logging if you want to dig deeper into the requests you're making against the service.

See our troubleshooting guide for details on how to diagnose various failure scenarios.

Next steps

- Go further with Azure.Search.Documents and our samples
- Watch a demo or deep dive video
- Read more about the Azure Cognitive Search service

Contributing

See our Search CONTRIBUTING.md for details on building, testing, and contributing to this library.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit cla.microsoft.com .

This project has adopted the Microsoft Open Source Code of Conduct . For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with

any additional questions or comments.

Microsoft.Azure.Management.Search Namespace

Reference

Classes

AdminKeysOperationsExtensions	Extension methods for AdminKeysOperations.
OperationsExtensions	Extension methods for Operations.
PrivateEndpointConnectionsOperationsExtensions	Extension methods for PrivateEndpointConnectionsOperations.
PrivateLinkResourcesOperationsExtensions	Extension methods for PrivateLinkResourcesOperations.
QueryKeysOperationsExtensions	Extension methods for QueryKeysOperations.
SearchManagementClient	Client that can be used to manage Azure Cognitive Search services and API keys.
ServicesOperationsExtensions	Extension methods for ServicesOperations.
SharedPrivateLinkResourcesOperationsExtensions	Extension methods for SharedPrivateLinkResourcesOperations.

Interfaces

IAdminKeysOperations	AdminKeysOperations operations.
IOperations	Operations operations.
IPrivateEndpointConnectionsOperations	PrivateEndpointConnectionsOperations operations.
IPrivateLinkResourcesOperations	PrivateLinkResourcesOperations operations.
IQueryKeysOperations	QueryKeysOperations operations.
ISearchManagementClient	Client that can be used to manage Azure Cognitive Search services and API keys.

IServicesOperations	ServicesOperations operations.
ISharedPrivateLinkResources Operations	SharedPrivateLinkResourcesOperations operations.

Azure Cognitive Search client library for Java - version 11.5.7

Article • 05/23/2023

This is the Java client library for Azure Cognitive Search. Azure Cognitive Search service is a search-as-a-service cloud solution that gives developers APIs and tools for adding a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

The Azure Cognitive Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and large text documents. A skillset leverages AI from Cognitive Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines.

Use the Azure Cognitive Search client library to:

- Submit queries for simple and advanced query forms that include fuzzy search, wildcard search, regular expressions.
- Implement filtered queries for faceted navigation, geospatial search, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through scoring profiles to factor in business logic or freshness.

Source code | Package (Maven) | API reference documentation | Product documentation | Samples

Getting started

Include the package

```XML
<dependency>
    <groupId>com.azure</groupId>
    <artifactId>azure-search-documents</artifactId>
    <version>11.5.7</version>
</dependency>
```

Prerequisites

- Java Development Kit (JDK) with version 8 or above
- Azure subscription
- Azure Cognitive Search service
- To create a new search service, you can use the Azure portal, Azure PowerShell, or the Azure CLI. Here's an example using the Azure CLI to create a free instance for getting started:

```Bash
az search service create --name <mysearch> --resource-group <mysearch-rg> --
sku free --location westus
```

See choosing a pricing tier for more information about available options.

Authenticate the client

In order to interact with the Azure Cognitive Search service you'll need to create an instance of the Search Client class. To make this possible you will need,

1. URL endpoint
2. API key

for your service. The api-key is the sole mechanism for authenticating access to your search service endpoint. You can obtain your api-key from the Azure portal or via the Azure CLI:

```Bash
az search admin-key show --service-name <mysearch> --resource-group
<mysearch-rg>
```

Note:

- The example Azure CLI snippet above retrieves an admin key. This allows for easier access when exploring APIs, but it should be managed carefully.
- There are two types of keys used to access your search service: **admin** *(read-write)* and **query** *(read-only)* keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

The SDK provides three clients.

- `SearchIndexClient` for CRUD operations on indexes and synonym maps.
- `SearchIndexerClient` for CRUD operations on indexers, date sources, and skillsets.
- `SearchClient` for all document operations.

Create a SearchIndexClient

To create a `SearchIndexClient/SearchIndexAsyncClient`, you will need the values of the Azure Cognitive Search service URL endpoint and admin key.

Java

```
SearchIndexClient searchIndexClient = new SearchIndexClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(apiKey))
    .buildClient();
```

or

Java

```
SearchIndexAsyncClient searchIndexAsyncClient = new
SearchIndexClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(apiKey))
    .buildAsyncClient();
```

Create a SearchIndexerClient

To create a `SearchIndexerClient/SearchIndexerAsyncClient`, you will need the values of the Azure Cognitive Search service URL endpoint and admin key.

Java

```
SearchIndexerClient searchIndexerClient = new SearchIndexerClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(apiKey))
    .buildClient();
```

or

Java

```
SearchIndexerAsyncClient searchIndexerAsyncClient = new
SearchIndexerClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(apiKey))
    .buildAsyncClient();
```

Create a SearchClient

Once you have the values of the Azure Cognitive Search service URL endpoint and admin key, you can create the `SearchClient/SearchAsyncClient` with an existing index name:

Java

```
SearchClient searchClient = new SearchClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(adminKey))
    .indexName(indexName)
    .buildClient();
```

or

Java

```
SearchAsyncClient searchAsyncClient = new SearchClientBuilder()
    .endpoint(endpoint)
    .credential(new AzureKeyCredential(adminKey))
    .indexName(indexName)
    .buildAsyncClient();
```

Send your first search query

To get running with Azure Cognitive Search first create an index following this guide. With an index created you can use the following samples to begin using the SDK.

Key concepts

An Azure Cognitive Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. *(If you're new to search, you can make a very rough analogy between indexes and database tables.)* The `azure-search-documents` client library exposes operations on these resources through two main client types.

- `SearchClient` helps with:
  - Searching your indexed documents using rich queries and powerful data shaping
  - Autocompleting partially typed search terms based on documents in the index
  - Suggesting the most likely matching text in documents as a user types
  - Adding, Updating or Deleting Documents documents from an index

- `SearchIndexClient` allows you to:
  - Create, delete, update, or configure a search index
  - Declare custom synonym maps to expand or rewrite queries
  - Most of the `SearchServiceClient` functionality is not yet available in our current preview

- `SearchIndexerClient` allows you to:
  - Start indexers to automatically crawl data sources
  - Define AI powered Skillsets to transform and enrich your data

Examples

The following examples all use a simple Hotel data set that you can import into your own index from the Azure portal. These are just a few of the basics - please check out our Samples for much more.

- Querying
  - Use SearchDocument like a dictionary for search results
  - Use Java model for search results
  - Search Options
- Creating an index
- Adding documents to your index
- Retrieving a specific document from your index
- Async APIs

- Create a client that can authenticate in a national cloud

Querying

There are two ways to interact with the data returned from a search query.

Let's explore them with a search for a "luxury" hotel.

Use `SearchDocument` like a dictionary for search results

`SearchDocument` is the default type returned from queries when you don't provide your own. Here we perform the search, enumerate over the results, and extract data using `SearchDocument`'s dictionary indexer.

Java

```
for (SearchResult searchResult : searchClient.search("luxury")) {
    SearchDocument doc = searchResult.getDocument(SearchDocument.class);
    String id = (String) doc.get("hotelId");
    String name = (String) doc.get("hotelName");
    System.out.printf("This is hotelId %s, and this is hotel name %s.%n",
id, name);
}
```

Use Java model class for search results

Define a `Hotel` class.

Java

```
public class Hotel {
    private String id;
    private String name;

    public String getId() {
        return id;
    }

    public Hotel setId(String id) {
        this.id = id;
        return this;
    }

    public String getName() {
        return name;
    }

    public Hotel setName(String name) {
        this.name = name;
        return this;
```

```
    }
}
```

Use it in place of `SearchDocument` when querying.

```Java
for (SearchResult searchResult : searchClient.search("luxury")) {
    Hotel doc = searchResult.getDocument(Hotel.class);
    String id = doc.getId();
    String name = doc.getName();
    System.out.printf("This is hotelId %s, and this is hotel name %s.%n",
id, name);
}
```

It is recommended, when you know the schema of the search index, to create a Java model class.

Search Options

The `SearchOptions` provide powerful control over the behavior of our queries.

Let's search for the top 5 luxury hotels with a good rating.

```Java
SearchOptions options = new SearchOptions()
    .setFilter("rating ge 4")
    .setOrderBy("rating desc")
    .setTop(5);
SearchPagedIterable searchResultsIterable = searchClient.search("luxury",
options, Context.NONE);
// ...
```

Creating an index

You can use the SearchIndexClient to create a search index. Indexes can also define suggesters, lexical analyzers, and more.

There are multiple ways of preparing search fields for a search index. For basic needs, we provide a static helper method `buildSearchFields` in `SearchIndexClient` and `SearchIndexAsyncClient`, which can convert Java POJO class into `List<SearchField>`. There are three annotations `SimpleFieldProperty`, `SearchFieldProperty` and `FieldBuilderIgnore` to configure the field of model class.

Java

```
List<SearchField> searchFields =
SearchIndexClient.buildSearchFields(Hotel.class, null);
searchIndexClient.createIndex(new SearchIndex("index", searchFields));
```

For advanced scenarios, we can build search fields using `SearchField` directly.

Java

```
List<SearchField> searchFieldList = new ArrayList<>();
searchFieldList.add(new SearchField("hotelId", SearchFieldDataType.STRING)
    .setKey(true)
    .setFilterable(true)
    .setSortable(true));

searchFieldList.add(new SearchField("hotelName", SearchFieldDataType.STRING)
    .setSearchable(true)
    .setFilterable(true)
    .setSortable(true));
searchFieldList.add(new SearchField("description",
SearchFieldDataType.STRING)
    .setSearchable(true)
    .setAnalyzerName(LexicalAnalyzerName.EU_LUCENE));
searchFieldList.add(new SearchField("tags",
SearchFieldDataType.collection(SearchFieldDataType.STRING))
    .setSearchable(true)
    .setFilterable(true)
    .setFacetable(true));
searchFieldList.add(new SearchField("address", SearchFieldDataType.COMPLEX)
    .setFields(new SearchField("streetAddress",
SearchFieldDataType.STRING).setSearchable(true),
        new SearchField("city", SearchFieldDataType.STRING)
            .setSearchable(true)
            .setFilterable(true)
            .setFacetable(true)
            .setSortable(true),
        new SearchField("stateProvince", SearchFieldDataType.STRING)
            .setSearchable(true)
            .setFilterable(true)
            .setFacetable(true)
            .setSortable(true),
        new SearchField("country", SearchFieldDataType.STRING)
            .setSearchable(true)
            .setFilterable(true)
            .setFacetable(true)
            .setSortable(true),
        new SearchField("postalCode", SearchFieldDataType.STRING)
            .setSearchable(true)
            .setFilterable(true)
            .setFacetable(true)
            .setSortable(true)
    ));
```

```
// Prepare suggester.
SearchSuggester suggester = new SearchSuggester("sg",
Collections.singletonList("hotelName"));
// Prepare SearchIndex with index name and search fields.
SearchIndex index = new
SearchIndex("hotels").setFields(searchFieldList).setSuggesters(suggester);
// Create an index
searchIndexClient.createIndex(index);
```

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

Java

```
Hotel hotel = searchClient.getDocument("1", Hotel.class);
System.out.printf("This is hotelId %s, and this is hotel name %s.%n",
hotel.getId(), hotel.getName());
```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are a few special rules for merging to be aware of.

Java

```
IndexDocumentsBatch<Hotel> batch = new IndexDocumentsBatch<>();
batch.addUploadActions(Collections.singletonList(new
Hotel().setId("783").setName("Upload Inn")));
batch.addMergeActions(Collections.singletonList(new
Hotel().setId("12").setName("Renovated Ranch")));
searchClient.indexDocuments(batch);
```

The request will throw `IndexBatchException` by default if any of the individual actions fail, and you can use `findFailedActionsToRetry` to retry on failed documents. There's also a `throwOnAnyError` option, and you can set it to `false` to get a successful response with an `IndexDocumentsResult` for inspection.

Async APIs

The examples so far have been using synchronous APIs, but we provide full support for async APIs as well. You'll need to use SearchAsyncClient.

Java

```
searchAsyncClient.search("luxury")
    .subscribe(result -> {
        Hotel hotel = result.getDocument(Hotel.class);
        System.out.printf("This is hotelId %s, and this is hotel name
%s.%n", hotel.getId(), hotel.getName());
    });
```

Authenticate in a National Cloud

To authenticate in a National Cloud, you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `audience` in `SearchClientBuilder`, `SearchIndexClientBuilder`, or `SearchIndexerClientBuilder`

Java

```
// Create a SearchClient that will authenticate through AAD in the China
national cloud.
SearchClient searchClient = new SearchClientBuilder()
    .endpoint(endpoint)
    .indexName(indexName)
    .credential(new DefaultAzureCredentialBuilder()
        .authorityHost(AzureAuthorityHosts.AZURE_CHINA)
        .build())
    .audience(SearchAudience.AZURE_CHINA)
    .buildClient();
```

Troubleshooting

General

When you interact with Azure Cognitive Search using this Java client library, errors returned by the service correspond to the same HTTP status codes returned for REST API requests. For example, the service will return a `404` error if you try to retrieve a document that doesn't exist in your index.

Handling Search Error Response

Any Search API operation that fails will throw an HttpResponseException with helpful Status codes. Many of these errors are recoverable.

Java

```
try {
    Iterable<SearchResult> results = searchClient.search("hotel");
} catch (HttpResponseException ex) {
    // The exception contains the HTTP status code and the detailed message
    // returned from the search service
    HttpResponse response = ex.getResponse();
    System.out.println("Status Code: " + response.getStatusCode());
    System.out.println("Message: " + ex.getMessage());
}
```

You can also easily enable console logging if you want to dig deeper into the requests you're making against the service.

Enabling Logging

Azure SDKs for Java provide a consistent logging story to help aid in troubleshooting application errors and expedite their resolution. The logs produced will capture the flow of an application before reaching the terminal state to help locate the root issue. View the logging wiki for guidance about enabling logging.

Default HTTP Client

By default, a Netty based HTTP client will be used. The HTTP clients wiki provides more information on configuring or changing the HTTP client.

Next steps

- Samples are explained in detail here.
- Watch a demo or deep dive video
- Read more about the Azure Cognitive Search service

Contributing

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to,

and actually do, grant us the rights to use your contribution.

When you submit a pull request, a CLA-bot will automatically determine whether you need to provide a CLA and decorate the PR appropriately (e.g., label, comment). Simply follow the instructions provided by the bot. You will only need to do this once across all repos using our CLA.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Management

Reference

Packages

com.microsoft.azure.management.search

Azure Cognitive Search client library for JavaScript - Version 11.3.1

Article • 04/04/2023

Azure Cognitive Search is a search-as-a-service cloud solution that gives developers APIs and tools for adding a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

The Azure Cognitive Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and large text documents. A skillset leverages AI from Cognitive Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines.

Use the @azure/search-documents client library to:

- Submit queries for simple and advanced query forms that include fuzzy search, wildcard search, regular expressions.
- Implement filtered queries for faceted navigation, geospatial search, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through scoring profiles to factor in business logic or freshness.

Source code | Package (NPM) | API reference documentation | REST API documentation | Product documentation | Samples

Getting started

Install the `@azure/search-documents` package

```bash
npm install @azure/search-documents
```

Currently supported environments

- LTS versions of Node.js
- Latest versions of Safari, Chrome, Edge, and Firefox.

See our support policy for more details.

Prerequisites

- An Azure subscription
- A Search service

To create a new search service, you can use the Azure portal, Azure PowerShell, or the Azure CLI. Here's an example using the Azure CLI to create a free instance for getting started:

```powershell
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free --location westus
```

See choosing a pricing tier for more information about available options.

Authenticate the client

All requests to a search service need an API key that was generated specifically for your service. The API key is the sole mechanism for authenticating access to your search service endpoint.

You can obtain your api-key from the Azure portal or via the Azure CLI:

```powershell
az search admin-key show --resource-group <your-resource-group-name> --service-name <your-resource-name>
```

Alternatively, you can get the endpoint and Admin Key from the resource information in the Azure Portal.

There are two types of keys used to access your search service: **admin** *(read-write)* and **query** *(read-only)* keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

Once you have an api-key, you can use it as follows:

JavaScript

```javascript
const {
  SearchClient,
  SearchIndexClient,
  SearchIndexerClient,
  AzureKeyCredential,
} = require("@azure/search-documents");

// To query and manipulate documents
const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

// To manage indexes and synonymmaps
const indexClient = new SearchIndexClient("<endpoint>", new
AzureKeyCredential("<apiKey>"));

// To manage indexers, datasources and skillsets
const indexerClient = new SearchIndexerClient("<endpoint>", new
AzureKeyCredential("<apiKey>"));
```

Authenticate in a National Cloud

To authenticate in a National Cloud, you will need to make the following additions to your client configuration:

- Set the `Audience` in `SearchClientOptions`

JavaScript

```javascript
const {
  SearchClient,
  SearchIndexClient,
  SearchIndexerClient,
  AzureKeyCredential,
  KnownSearchAudience,
```

```
} = require("@azure/search-documents");

// To query and manipulate documents
const searchClient = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>"),
  {
    audience: KnownSearchAudience.AzureChina,
  }
);

// To manage indexes and synonymmaps
const indexClient = new SearchIndexClient("<endpoint>", new
AzureKeyCredential("<apiKey>"), {
  audience: KnownSearchAudience.AzureChina,
});

// To manage indexers, datasources and skillsets
const indexerClient = new SearchIndexerClient("<endpoint>", new
AzureKeyCredential("<apiKey>"), {
  audience: KnownSearchAudience.AzureChina,
});
```

Key concepts

An Azure Cognitive Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. *(If you're brand new to search, you can make a very rough analogy between indexes and database tables.)* The @azure/search-documents client library exposes operations on these resources through three main client types.

- `SearchClient` helps with:
  - Searching your indexed documents using rich queries and powerful data shaping
  - Autocompleting partially typed search terms based on documents in the index
  - Suggesting the most likely matching text in documents as a user types
  - Adding, Updating or Deleting Documents documents from an index
- `SearchIndexClient` allows you to:
  - Create, delete, update, or configure a search index
  - Declare custom synonym maps to expand or rewrite queries
- `SearchIndexerClient` allows you to:
  - Start indexers to automatically crawl data sources
  - Define AI powered Skillsets to transform and enrich your data

Note: These clients cannot function in the browser because the APIs it calls do not have support for Cross-Origin Resource Sharing (CORS).

TypeScript/JavaScript specific concepts

Documents

An item stored inside a search index. The shape of this document is described in the index using `Field`s. Each Field has a name, a datatype, and additional metadata such as if it is searchable or filterable.

Pagination

Typically you will only wish to show a subset of search results to a user at one time. To support this, you can use the `top`, `skip` and `includeTotalCount` parameters to provide a paged experience on top of search results.

Document field encoding

Supported data types in an index are mapped to JSON types in API requests/responses. The JS client library keeps these mostly the same, with some exceptions:

- `Edm.DateTimeOffset` is converted to a JS `Date`.
- `Edm.GeographyPoint` is converted to a `GeographyPoint` type exported by the client library.
- Special values of the `number` type (NaN, Infinity, -Infinity) are serialized as strings in the REST API, but are converted back to `number` by the client library.

Note: Data types are converted based on value, not the field type in the index schema. This means that if you have an ISO8601 Date string (e.g. "2020-03-06T18:48:27.896Z") as the value of a field, it will be converted to a Date regardless of how you stored it in your schema.

Examples

The following examples demonstrate the basics - please check out our samples for much more.

- Creating an index
- Retrieving a specific document from your index

- Adding documents to your index
- Perform a search on documents
  - Querying with TypeScript
  - Querying with OData filters
  - Querying with facets

Create an Index

JavaScript

```
const { SearchIndexClient, AzureKeyCredential } = require("@azure/search-documents");

const client = new SearchIndexClient("<endpoint>", new AzureKeyCredential("<apiKey>"));

async function main() {
  const result = await client.createIndex({
    name: "example-index",
    fields: [
      {
        type: "Edm.String",
        name: "id",
        key: true,
      },
      {
        type: "Edm.Double",
        name: "awesomenessLevel",
        sortable: true,
        filterable: true,
        facetable: true,
      },
      {
        type: "Edm.String",
        name: "description",
        searchable: true,
      },
      {
        type: "Edm.ComplexType",
        name: "details",
        fields: [
          {
            type: "Collection(Edm.String)",
            name: "tags",
            searchable: true,
          },
        ],
      },
      {
        type: "Edm.Int32",
        name: "hiddenWeight",
```

```
        hidden: true,
      },
    ],
  });

  console.log(result);
}

main();
```

Retrieve a specific document from an index

A specific document can be retrieved by its primary key value:

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");

const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const result = await client.getDocument("1234");
  console.log(result);
}

main();
```

Adding documents into an index

You can upload multiple documents into index inside a batch:

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");

const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const uploadResult = await client.uploadDocuments([
```

```
    // JSON objects matching the shape of the client's index
    {},
    {},
    {},
  ]);
  for (const result of uploadResult.results) {
    console.log(`Uploaded ${result.key}; succeeded? ${result.succeeded}`);
  }
}

main();
```

Perform a search on documents

To list all results of a particular query, you can use `search` with a search string that uses simple query syntax:

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");

const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const searchResults = await client.search("wifi -luxury");
  for await (const result of searchResults.results) {
    console.log(result);
  }
}

main();
```

For a more advanced search that uses Lucene syntax, specify `queryType` to be `full`:

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-
documents");

const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);
```

```
async function main() {
  const searchResults = await client.search('Category:budget AND "recently
renovated"^3', {
    queryType: "full",
    searchMode: "all",
  });
  for await (const result of searchResults.results) {
    console.log(result);
  }
}

main();
```

Querying with TypeScript

In TypeScript `SearchClient` takes a generic parameter that is the model shape of your index documents. This allows you to perform strongly typed lookup of fields returned in results. TypeScript is also able to check for fields returned when specifying a `select` parameter.

ts

```
import { SearchClient, AzureKeyCredential } from "@azure/search-documents";

// An example schema for documents in the index
interface Hotel {
  HotelId: string;
  HotelName: string;
  Description: string;
  ParkingIncluded: boolean;
  LastRenovationDate: Date;
  Rating: number;
}

const client = new SearchClient<Hotel>(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const searchResults = await client.search("wifi -luxury", {
    // Only fields in Hotel can be added to this array.
    // TS will complain if one is misspelled.
    select: ["HotelId", "HotelName", "Rating"],
  });

  for await (const result of searchResults.results) {
    // result.document has HotelId, HotelName, and Rating.
    // Trying to access result.document.Description would emit a TS error.
    console.log(result.document.HotelName);
```

```
  }
}

main();
```

Querying with OData filters

Using the `filter` query parameter allows you to query an index using the syntax of an OData $filter expression.

JavaScript

```
const { SearchClient, AzureKeyCredential, odata } = require("@azure/search-documents");

const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const baseRateMax = 200;
  const ratingMin = 4;
  const searchResults = await client.search("WiFi", {
    filter: odata`Rooms/any(room: room/BaseRate lt ${baseRateMax}) and Rating ge ${ratingMin}`,
    orderBy: ["Rating desc"],
    select: ["HotelId", "HotelName", "Rating"],
  });
  for await (const result of searchResults.results) {
    // Each result will have "HotelId", "HotelName", and "Rating"
    // in addition to the standard search result property "score"
    console.log(result);
  }
}

main();
```

Querying with facets

Facets are used to help a user of your application refine a search along pre-configured dimensions. Facet syntax provides the options to sort and bucket facet values.

JavaScript

```
const { SearchClient, AzureKeyCredential } = require("@azure/search-documents");
```

```
const client = new SearchClient(
  "<endpoint>",
  "<indexName>",
  new AzureKeyCredential("<apiKey>")
);

async function main() {
  const searchResults = await client.search("WiFi", {
    facets: ["Category,count:3,sort:count", "Rooms/BaseRate,interval:100"],
  });
  console.log(searchResults.facets);
  // Output will look like:
  // {
  //   'Rooms/BaseRate': [
  //     { count: 16, value: 0 },
  //     { count: 17, value: 100 },
  //     { count: 17, value: 200 }
  //   ],
  //   Category: [
  //     { count: 5, value: 'Budget' },
  //     { count: 5, value: 'Luxury' },
  //     { count: 5, value: 'Resort and Spa' }
  //   ]
  // }
}

main();
```

When retrieving results, a `facets` property will be available that will indicate the number of results that fall into each facet bucket. This can be used to drive refinement (e.g. issuing a follow-up search that filters on the `Rating` being greater than or equal to 3 and less than 4.)

Troubleshooting

Logging

Enabling logging may help uncover useful information about failures. In order to see a log of HTTP requests and responses, set the `AZURE_LOG_LEVEL` environment variable to `info`. Alternatively, logging can be enabled at runtime by calling `setLogLevel` in the `@azure/logger`:

JavaScript

```
import { setLogLevel } from "@azure/logger";
```

```
setLogLevel("info");
```

For more detailed instructions on how to enable logs, you can look at the @azure/logger package docs .

Next steps

- Go further with search-documents and our samples
- Watch a demo or deep dive video
- Read more about the Azure Cognitive Search service

Contributing

If you'd like to contribute to this library, please read the contributing guide to learn more about how to build and test the code.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit cla.microsoft.com .

This project has adopted the Microsoft Open Source Code of Conduct .For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Related projects

- Microsoft Azure SDK for Javascript

@azure/arm-search package

Reference

Classes

SearchManagementClient

Interfaces

AdminKeyResult	Response containing the primary and secondary admin API keys for a given Azure Cognitive Search service.
AdminKeys	Interface representing a AdminKeys.
AdminKeysGetOptional Params	Optional parameters.
AdminKeysRegenerate OptionalParams	Optional parameters.
AsyncOperationResult	The details of a long running asynchronous shared private link resource operation
CheckNameAvailabilityInput	Input of check name availability API.
CheckNameAvailabilityOutput	Output of check name availability API.
CloudError	Contains information about an API error.
CloudErrorBody	Describes a particular API error with an error code and a message.
Identity	Identity for the resource.
IpRule	The IP restriction rule of the Azure Cognitive Search service.
ListQueryKeysResult	Response containing the query API keys for a given Azure Cognitive Search service.
NetworkRuleSet	Network specific rules that determine how the Azure Cognitive Search service may be reached.
Operation	Describes a REST API operation.
OperationDisplay	The object that describes the operation.
OperationListResult	The result of the request to list REST API operations. It contains a

	list of operations and a URL to get the next set of results.
Operations	Interface representing a Operations.
OperationsListOptionalParams	Optional parameters.
PrivateEndpointConnection	Describes an existing Private Endpoint connection to the Azure Cognitive Search service.
PrivateEndpointConnection ListResult	Response containing a list of Private Endpoint connections.
PrivateEndpointConnection Properties	Describes the properties of an existing Private Endpoint connection to the Azure Cognitive Search service.
PrivateEndpointConnection PropertiesPrivateEndpoint	The private endpoint resource from Microsoft.Network provider.
PrivateEndpointConnection PropertiesPrivateLinkService ConnectionState	Describes the current state of an existing Private Link Service connection to the Azure Private Endpoint.
PrivateEndpointConnections	Interface representing a PrivateEndpointConnections.
PrivateEndpointConnections DeleteOptionalParams	Optional parameters.
PrivateEndpointConnections GetOptionalParams	Optional parameters.
PrivateEndpointConnections ListByServiceNextOptional Params	Optional parameters.
PrivateEndpointConnections ListByServiceOptionalParams	Optional parameters.
PrivateEndpointConnections UpdateOptionalParams	Optional parameters.
PrivateLinkResource	Describes a supported private link resource for the Azure Cognitive Search service.
PrivateLinkResourceProperties	Describes the properties of a supported private link resource for the Azure Cognitive Search service. For a given API version, this represents the 'supported' groupIds when creating a shared private link resource.
PrivateLinkResources	Interface representing a PrivateLinkResources.
PrivateLinkResourcesList SupportedOptionalParams	Optional parameters.

PrivateLinkResourcesResult	Response containing a list of supported Private Link Resources.
QueryKey	Describes an API key for a given Azure Cognitive Search service that has permissions for query operations only.
QueryKeys	Interface representing a QueryKeys.
QueryKeysCreateOptional Params	Optional parameters.
QueryKeysDeleteOptional Params	Optional parameters.
QueryKeysListBySearchService NextOptionalParams	Optional parameters.
QueryKeysListBySearchService OptionalParams	Optional parameters.
Resource	Common fields that are returned in the response for all Azure Resource Manager resources
SearchManagementClient OptionalParams	Optional parameters.
SearchManagementRequest Options	Parameter group
SearchService	Describes an Azure Cognitive Search service and its current state.
SearchServiceListResult	Response containing a list of Azure Cognitive Search services.
SearchServiceUpdate	The parameters used to update an Azure Cognitive Search service.
Services	Interface representing a Services.
ServicesCheckName AvailabilityOptionalParams	Optional parameters.
ServicesCreateOrUpdate OptionalParams	Optional parameters.
ServicesDeleteOptionalParams	Optional parameters.
ServicesGetOptionalParams	Optional parameters.
ServicesListByResourceGroup NextOptionalParams	Optional parameters.
ServicesListByResourceGroup OptionalParams	Optional parameters.

ServicesListBySubscriptionNextOptionalParams	Optional parameters.
ServicesListBySubscriptionOptionalParams	Optional parameters.
ServicesUpdateOptionalParams	Optional parameters.
ShareablePrivateLinkResourceProperties	Describes the properties of a resource type that has been onboarded to private link service, supported by Azure Cognitive Search.
ShareablePrivateLinkResourceType	Describes an resource type that has been onboarded to private link service, supported by Azure Cognitive Search.
SharedPrivateLinkResource	Describes a Shared Private Link Resource managed by the Azure Cognitive Search service.
SharedPrivateLinkResourceListResult	Response containing a list of Shared Private Link Resources.
SharedPrivateLinkResourceProperties	Describes the properties of an existing Shared Private Link Resource managed by the Azure Cognitive Search service.
SharedPrivateLinkResources	Interface representing a SharedPrivateLinkResources.
SharedPrivateLinkResourcesCreateOrUpdateOptionalParams	Optional parameters.
SharedPrivateLinkResourcesDeleteOptionalParams	Optional parameters.
SharedPrivateLinkResourcesGetOptionalParams	Optional parameters.
SharedPrivateLinkResourcesListByServiceNextOptionalParams	Optional parameters.
SharedPrivateLinkResourcesListByServiceOptionalParams	Optional parameters.
Sku	Defines the SKU of an Azure Cognitive Search Service, which determines price tier and capacity limits.
TrackedResource	The resource model definition for an Azure Resource Manager tracked top level resource which has 'tags' and a 'location'

Type Aliases

AdminKeyKind	Defines values for AdminKeyKind.
AdminKeysGetResponse	Contains response data for the get operation.
AdminKeysRegenerateResponse	Contains response data for the regenerate operation.
HostingMode	Defines values for HostingMode.
IdentityType	Defines values for IdentityType.
OperationsListResponse	Contains response data for the list operation.
PrivateEndpointConnectionsDeleteResponse	Contains response data for the delete operation.
PrivateEndpointConnectionsGetResponse	Contains response data for the get operation.
PrivateEndpointConnectionsListByServiceNextResponse	Contains response data for the listByServiceNext operation.
PrivateEndpointConnectionsListByServiceResponse	Contains response data for the listByService operation.
PrivateEndpointConnectionsUpdateResponse	Contains response data for the update operation.
PrivateLinkResourcesListSupportedResponse	Contains response data for the listSupported operation.
PrivateLinkServiceConnectionStatus	Defines values for PrivateLinkServiceConnectionStatus.
ProvisioningState	Defines values for ProvisioningState.
PublicNetworkAccess	Defines values for PublicNetworkAccess.
QueryKeysCreateResponse	Contains response data for the create operation.
QueryKeysListBySearchServiceNextResponse	Contains response data for the listBySearchServiceNext operation.
QueryKeysListBySearchServiceResponse	Contains response data for the listBySearchService operation.
SearchServiceStatus	Defines values for SearchServiceStatus.
ServicesCheckNameAvailabilityResponse	Contains response data for the checkNameAvailability operation.
ServicesCreateOrUpdateResponse	Contains response data for the createOrUpdate operation.

ServicesGetResponse	Contains response data for the get operation.
ServicesListByResourceGroupNextResponse	Contains response data for the listByResourceGroupNext operation.
ServicesListByResourceGroupResponse	Contains response data for the listByResourceGroup operation.
ServicesListBySubscriptionNextResponse	Contains response data for the listBySubscriptionNext operation.
ServicesListBySubscriptionResponse	Contains response data for the listBySubscription operation.
ServicesUpdateResponse	Contains response data for the update operation.
SharedPrivateLinkResourceAsyncOperationResult	Defines values for SharedPrivateLinkResourceAsyncOperationResult. KnownSharedPrivateLinkResourceAsyncOperationResult can be used interchangeably with SharedPrivateLinkResourceAsyncOperationResult, this enum contains the known values that the service supports. **Known values supported by the service** **Running** **Succeeded** **Failed**
SharedPrivateLinkResourceProvisioningState	Defines values for SharedPrivateLinkResourceProvisioningState.
SharedPrivateLinkResourceStatus	Defines values for SharedPrivateLinkResourceStatus.
SharedPrivateLinkResourcesCreateOrUpdateResponse	Contains response data for the createOrUpdate operation.
SharedPrivateLinkResourcesGetResponse	Contains response data for the get operation.
SharedPrivateLinkResourcesListByServiceNextResponse	Contains response data for the listByServiceNext operation.
SharedPrivateLinkResourcesListByServiceResponse	Contains response data for the listByService operation.
SkuName	Defines values for SkuName.

UnavailableNameReason	Defines values for UnavailableNameReason. KnownUnavailableNameReason can be used interchangeably with UnavailableNameReason, this enum contains the known values that the service supports. **Known values supported by the service** **Invalid** **AlreadyExists**

Enums

KnownSharedPrivateLinkResourceAsyncOperationResult	Known values of SharedPrivateLinkResourceAsyncOperationResult that the service accepts.
KnownUnavailableNameReason	Known values of UnavailableNameReason that the service accepts.

Functions

getContinuationToken(unknown)	Given the last `.value` produced by the `byPage` iterator, returns a continuation token that can be used to begin paging from that point later.

Function Details

getContinuationToken(unknown)

Given the last `.value` produced by the `byPage` iterator, returns a continuation token that can be used to begin paging from that point later.

TypeScript

```
function getContinuationToken(page: unknown): string | undefined
```

Parameters

page unknown

An object from accessing `value` on the IteratorResult from a `byPage` iterator.

Returns

string | undefined

The continuation token that can be passed into byPage() during future calls.

Azure Cognitive Search client library for Python - version 11.3.0

Article • 09/06/2022 • 7 minutes to read

Azure Cognitive Search is a search-as-a-service cloud solution that gives developers APIs and tools for adding a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

The Azure Cognitive Search service is well suited for the following application scenarios:

- Consolidate varied content types into a single searchable index. To populate an index, you can push JSON documents that contain your content, or if your data is already in Azure, create an indexer to pull in data automatically.
- Attach skillsets to an indexer to create searchable content from images and large text documents. A skillset leverages AI from Cognitive Services for built-in OCR, entity recognition, key phrase extraction, language detection, text translation, and sentiment analysis. You can also add custom skills to integrate external processing of your content during data ingestion.
- In a search client application, implement query logic and user experiences similar to commercial web search engines.

Use the Azure.Search.Documents client library to:

- Submit queries for simple and advanced query forms that include fuzzy search, wildcard search, regular expressions.
- Implement filtered queries for faceted navigation, geospatial search, or to narrow results based on filter criteria.
- Create and manage search indexes.
- Upload and update documents in the search index.
- Create and manage indexers that pull data from Azure into an index.
- Create and manage skillsets that add AI enrichment to data ingestion.
- Create and manage analyzers for advanced text analysis or multi-lingual content.
- Optimize results through scoring profiles to factor in business logic or freshness.

Source code | Package (PyPI) | API reference documentation | Product documentation | Samples

Disclaimer

Azure SDK Python packages support for Python 2.7 has ended 01 January 2022. For more information and questions, please refer to https://github.com/Azure/azure-sdk-for-python/issues/20691

Getting started

Install the package

Install the Azure Cognitive Search client library for Python with pip:

```bash
pip install azure-search-documents
```

Prerequisites

- Python 3.7 or later is required to use this package.
- You need an Azure subscription and a Azure Cognitive Search service to use this package.

To create a new search service, you can use the Azure portal, Azure PowerShell, or the Azure CLI.

```powershell
az search service create --name <mysearch> --resource-group <mysearch-rg> --sku free --location westus
```

See choosing a pricing tier for more information about available options.

Authenticate the client

All requests to a search service need an api-key that was generated specifically for your service. The api-key is the sole mechanism for authenticating access to your search service endpoint. You can obtain your api-key from the Azure portal or via the Azure CLI:

```powershell
az search admin-key show --service-name <mysearch> --resource-group <mysearch-rg>
```

There are two types of keys used to access your search service: **admin** *(read-write)* and **query** *(read-only)* keys. Restricting access and operations in client apps is essential to safeguarding the search assets on your service. Always use a query key rather than an admin key for any query originating from a client app.

Note: The example Azure CLI snippet above retrieves an admin key so it's easier to get started exploring APIs, but it should be managed carefully.

We can use the api-key to create a new `SearchClient`.

Python

```python
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

index_name = "nycjobs"
# Get the service endpoint and API key from the environment
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

# Create a client
credential = AzureKeyCredential(key)
client = SearchClient(endpoint=endpoint,
                      index_name=index_name,
                      credential=credential)
```

Key concepts

An Azure Cognitive Search service contains one or more indexes that provide persistent storage of searchable data in the form of JSON documents. *(If you're brand new to search, you can make a very rough analogy between indexes and database tables.)* The Azure.Search.Documents client library exposes operations on these resources through two main client types.

- `SearchClient` helps with:
  - Searching your indexed documents using rich queries and powerful data shaping
  - Autocompleting partially typed search terms based on documents in the index
  - Suggesting the most likely matching text in documents as a user types
  - Adding, Updating or Deleting Documents documents from an index

- `SearchIndexClient` allows you to:
  - Create, delete, update, or configure a search index
  - Declare custom synonym maps to expand or rewrite queries

- Most of the `SearchServiceClient` functionality is not yet available in our current preview

- `SearchIndexerClient` allows you to:
  - Start indexers to automatically crawl data sources
  - Define AI powered Skillsets to transform and enrich your data

The `Azure.Search.Documents` client library (v1) is a brand new offering for Python developers who want to use search technology in their applications. There is an older, fully featured `Microsoft.Azure.Search` client library (v10) with many similar looking APIs, so please be careful to avoid confusion when exploring online resources.

Examples

The following examples all use a simple Hotel data set that you can import into your own index from the Azure portal. These are just a few of the basics - please check out our Samples for much more.

- Querying
- Creating an index
- Adding documents to your index
- Retrieving a specific document from your index
- Async APIs

Querying

Let's start by importing our namespaces.

Python

```python
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient
```

We'll then create a `SearchClient` to access our hotels search index.

Python

```python
index_name = "hotels"
# Get the service endpoint and API key from the environment
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

# Create a client
```

```
credential = AzureKeyCredential(key)
client = SearchClient(endpoint=endpoint,
                      index_name=index_name,
                      credential=credential)
```

Let's search for a "luxury" hotel.

Python

```python
results = client.search(search_text="luxury")

for result in results:
    print("{}: {})".format(result["hotelId"], result["hotelName"]))
```

Creating an index

You can use the `SearchIndexClient` to create a search index. Fields can be defined using convenient `SimpleField`, `SearchableField`, or `ComplexField` models. Indexes can also define suggesters, lexical analyzers, and more.

Python

```python
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import (
    ComplexField,
    CorsOptions,
    SearchIndex,
    ScoringProfile,
    SearchFieldDataType,
    SimpleField,
    SearchableField
)

endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

# Create a service client
client = SearchIndexClient(endpoint, AzureKeyCredential(key))

# Create the index
name = "hotels"
fields = [
        SimpleField(name="hotelId", type=SearchFieldDataType.String,
key=True),
        SimpleField(name="baseRate", type=SearchFieldDataType.Double),
        SearchableField(name="description",
type=SearchFieldDataType.String),
```

```
        ComplexField(name="address", fields=[
            SimpleField(name="streetAddress",
type=SearchFieldDataType.String),
            SimpleField(name="city", type=SearchFieldDataType.String),
        ])
    ]
cors_options = CorsOptions(allowed_origins=["*"], max_age_in_seconds=60)
scoring_profiles = []

index = SearchIndex(
    name=name,
    fields=fields,
    scoring_profiles=scoring_profiles,
    cors_options=cors_options)

result = client.create_index(index)
```

Adding documents to your index

You can `Upload`, `Merge`, `MergeOrUpload`, and `Delete` multiple documents from an index in a single batched request. There are a few special rules for merging to be aware of.

Python

```python
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

index_name = "hotels"
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

DOCUMENT = {
    'Category': 'Hotel',
    'hotelId': '1000',
    'rating': 4.0,
    'rooms': [],
    'hotelName': 'Azure Inn',
}

search_client = SearchClient(endpoint, index_name, AzureKeyCredential(key))

result = search_client.upload_documents(documents=[DOCUMENT])

print("Upload of new document succeeded: {}".format(result[0].succeeded))
```

Authenticate in a National Cloud

To authenticate in a National Cloud, you will need to make the following additions to your client configuration:

- Set the `AuthorityHost` in the credential options or via the `AZURE_AUTHORITY_HOST` environment variable
- Set the `audience` in `SearchClient`, `SearchIndexClient`, or `SearchIndexerClient`

Python

```
# Create a SearchClient that will authenticate through AAD in the China
national cloud.
import os
from azure.identity import DefaultAzureCredential, AzureAuthorityHosts
from azure.search.documents import SearchClient

index_name = "hotels"
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]
credential =
DefaultAzureCredential(authority=AzureAuthorityHosts.AZURE_CHINA)

search_client = SearchClient(endpoint, index_name, crdential=credential,
audience="https://search.azure.cn")
```

Retrieving a specific document from your index

In addition to querying for documents using keywords and optional filters, you can retrieve a specific document from your index if you already know the key. You could get the key from a query, for example, and want to show more information about it or navigate your customer to that document.

Python

```
import os
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

index_name = "hotels"
endpoint = os.environ["SEARCH_ENDPOINT"]
key = os.environ["SEARCH_API_KEY"]

client = SearchClient(endpoint, index_name, AzureKeyCredential(key))

result = client.get_document(key="1")

print("Details for hotel '1' are:")
print("        Name: {}".format(result["HotelName"]))
print("      Rating: {}".format(result["Rating"]))
print("    Category: {}".format(result["Category"]))
```

Async APIs

This library includes a complete async API. To use it, you must first install an async transport, such as aiohttp. See azure-core documentation for more information.

```py
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.aio import SearchClient

client = SearchClient(endpoint, index_name, AzureKeyCredential(api_key))

async with client:
  results = await client.search(search_text="hotel")
  async for result in results:
    print("{}: {})".format(result["hotelId"], result["hotelName"]))

...
```

Troubleshooting

General

The Azure Cognitive Search client will raise exceptions defined in Azure Core.

Logging

This library uses the standard logging library for logging. Basic information about HTTP sessions (URLs, headers, etc.) is logged at INFO level.

Detailed DEBUG level logging, including request/response bodies and unredacted headers, can be enabled on a client with the `logging_enable` keyword argument:

```Python
import sys
import logging
from azure.core.credentials import AzureKeyCredential
from azure.search.documents import SearchClient

# Create a logger for the 'azure' SDK
logger = logging.getLogger('azure')
logger.setLevel(logging.DEBUG)
```

```
# Configure a console output
handler = logging.StreamHandler(stream=sys.stdout)
logger.addHandler(handler)

# This client will log detailed information about its HTTP sessions, at
DEBUG level
client = SearchClient("<service endpoint>", "<index_name>",
AzureKeyCredential("<api key>"), logging_enable=True)
```

Similarly, `logging_enable` can enable detailed logging for a single operation, even when it isn't enabled for the client:

Python

```
result =  client.search(search_text="spa", logging_enable=True)
```

Next steps

- Go further with Azure.Search.Documents and our https://github.com/Azure/azure-sdk-for-python/blob/azure-search-documents_11.3.0/sdk/search/azure-search-documents/samples
- Watch a demo or deep dive video
- Read more about the Azure Cognitive Search service

Contributing

See our Search CONTRIBUTING.md for details on building, testing, and contributing to this library.

This project welcomes contributions and suggestions. Most contributions require you to agree to a Contributor License Agreement (CLA) declaring that you have the right to, and actually do, grant us the rights to use your contribution. For details, visit cla.microsoft.com.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

Related projects

- Microsoft Azure SDK for Python

azure-mgmt-search Package

Reference

Packages

search

az search

Reference

Manage Azure Search services, admin keys and query keys.

Commands

az search admin-key	Manage Azure Search admin keys.
az search admin-key renew	Regenerates either the primary or secondary admin API key.
az search admin-key show	Gets the primary and secondary admin API keys for the specified Azure Cognitive Search service.
az search private-endpoint-connection	Manage Azure Search private endpoint connections.
az search private-endpoint-connection delete	Disconnects the private endpoint connection and deletes it from the search service.
az search private-endpoint-connection list	Gets a list of all private endpoint connections in the given service.
az search private-endpoint-connection show	Gets the details of the private endpoint connection to the search service in the given resource group.
az search private-endpoint-connection update	Update an existing private endpoint connection in a Search service in the given resource group.
az search private-link-resource	Manage Azure Search private link resources.
az search private-link-resource list	Gets a list of all supported private link resource types for the given service.
az search query-key	Manage Azure Search query keys.
az search query-key create	Generates a new query key for the specified search service.
az search query-key delete	Deletes the specified query key.
az search query-key list	Returns the list of query API keys for the given Azure Cognitive Search service.
az search service	Manage Azure Search services.
az search service create	Creates a Search service in the given resource group.
az search service delete	Deletes a search service in the given resource group, along with

	its associated resources.
az search service list	Gets a list of all Search services in the given resource group.
az search service show	Gets the search service with the given name in the given resource group.
az search service update	Update partition and replica of the given search service.
az search service wait	Wait for async service operations.
az search shared-private-link-resource	Manage Azure Search shared private link resources.
az search shared-private-link-resource create	Create shared privatelink resources in a Search service in the given resource group.
az search shared-private-link-resource delete	Initiates the deletion of the shared private link resource from the search service.
az search shared-private-link-resource list	Gets a list of all shared private link resources managed by the given service.
az search shared-private-link-resource show	Gets the details of the shared private link resource managed by the search service in the given resource group.
az search shared-private-link-resource update	Update shared privatelink resources in a Search service in the given resource group.
az search shared-private-link-resource wait	Wait for async shared private link resource operations.

Az.Search

Reference

This topic displays help topics for the Azure Cognitive Search Cmdlets.

Search

Get-AzSearchAdminKeyPair	Gets admin key pair of the Azure Cognitive Search service.
Get-AzSearchPrivateEndpointConnection	Gets private endpoint connection(s) to the Azure Cognitive Search service.
Get-AzSearchPrivateLinkResource	Gets private link resource details for the Azure Cognitive Search service.
Get-AzSearchQueryKey	Gets query key(s) of the Azure Cognitive Search service.
Get-AzSearchService	Gets an Azure Cognitive Search service.
Get-AzSearchSharedPrivateLinkResource	Gets shared private link resources(s) of the Azure Cognitive Search service.
New-AzSearchAdminKey	Regenerates an admin key of the Azure Cognitive Search service.
New-AzSearchQueryKey	Create a new query key for the Azure Cognitive Search service.
New-AzSearchService	Creates an Azure Cognitive Search service.
New-AzSearchSharedPrivateLinkResource	Creates a shared private link resource for the Azure Cognitive Search service.
Remove-AzSearchPrivateEndpointConnection	Remove the private endpoint connection from the Azure Cognitive Search service.
Remove-AzSearchQueryKey	Remove the query key from the Azure Cognitive Search service.
Remove-AzSearchService	Remove an Azure Cognitive Search service.
Remove-AzSearchSharedPrivateLinkResource	Remove the shared private link resource from the Azure Cognitive Search service.
Set-AzSearchPrivateEndpointConnection	Update the private endpoint connection to the Azure Cognitive Search service.
Set-AzSearchService	Update an Azure Cognitive Search service.

Set-AzSearchSharedPrivateLinkResource	Update the shared private link resource for the Azure Cognitive Search service.

Microsoft.Search searchServices

Article • 12/28/2022 • 14 minutes to read

Bicep resource definition

The searchServices resource type can be deployed to:

- **Resource groups** - See resource group deployment commands

For a list of changed properties in each API version, see change log.

Resource format

To create a Microsoft.Search/searchServices resource, add the following Bicep to your template.

Bicep

```
resource symbolicname 'Microsoft.Search/searchServices@2021-04-01-preview' = {
  name: 'string'
  location: 'string'
  tags: {
    tagName1: 'tagValue1'
    tagName2: 'tagValue2'
  }
  sku: {
    name: 'string'
  }
  identity: {
    type: 'string'
    userAssignedIdentities: {}
  }
  properties: {
    authOptions: {
      aadOrApiKey: {
        aadAuthFailureMode: 'string'
      }
      apiKeyOnly: any()
    }
    disabledDataExfiltrationOptions: 'All'
    disableLocalAuth: bool
    encryptionWithCmk: {
      enforcement: 'string'
    }
    hostingMode: 'string'
    networkRuleSet: {
      bypass: 'string'
      ipRules: [
        {
          value: 'string'
        }
      ]
    }
    partitionCount: int
    publicNetworkAccess: 'string'
    replicaCount: int
    semanticSearch: 'string'
  }
}
```

Property values

searchServices

Name	Description	Value
name	The resource name	string (required)
location	The geo-location where the resource lives	string (required)
tags	Resource tags.	Dictionary of tag names and values. See Tags in templates

Name	Description	Value
sku	The SKU of the Search Service, which determines price tier and capacity limits. This property is required when creating a new Search Service.	Sku
identity	The identity of the resource.	Identity
properties	Properties of the search service.	SearchServiceProperties

Identity

Name	Description
type	The type of identity used for the resource. The type 'SystemAssigned, UserAssigned' includes both an identity created by the system and a set assigned identities. The type 'None' will remove all identities from the service.
userAssignedIdentities	The list of user identities associated with the resource. The user identity dictionary key references will be ARM resource ids in the form: '/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identity

SearchServiceProperties

Name	Description	Value
authOptions	Defines the options for how the data plane API of a search service authenticates requests. This cannot be set if 'disableLocalAuth' is set to true.	DataPlaneAuthOptions
disabledDataExfiltrationOptions	A list of data exfiltration scenarios that are explicitly disallowed for the search service. Currently, the only supported value is 'All' to disable all possible data export scenarios with more fine grained controls planned for the future.	String array containing any of: 'All'
disableLocalAuth	When set to true, calls to the search service will not be permitted to utilize API keys for authentication. This cannot be set to true if 'dataPlaneAuthOptions' are defined.	bool
encryptionWithCmk	Specifies any policy regarding encryption of resources (such as indexes) using customer manager keys within a search service.	EncryptionWithCmk
hostingMode	Applicable only for the standard3 SKU. You can set this property to enable up to 3 high density partitions that allow up to 1000 indexes, which is much higher than the maximum indexes allowed for any other SKU. For the standard3 SKU, the value is either 'default' or 'highDensity'. For all other SKUs, this value must be 'default'.	'default' 'highDensity'
networkRuleSet	Network specific rules that determine how the Azure Cognitive Search service may be reached.	NetworkRuleSet
partitionCount	The number of partitions in the search service; if specified, it can be 1, 2, 3, 4, 6, or 12. Values greater than 1 are only valid for standard SKUs. For 'standard3' services with hostingMode set to 'highDensity', the allowed values are between 1 and 3.	int
publicNetworkAccess	This value can be set to 'enabled' to avoid breaking changes on existing customer resources and templates. If set to 'disabled', traffic over public interface is not allowed, and private endpoint connections would be the exclusive access method.	'disabled' 'enabled'
replicaCount	The number of replicas in the search service. If specified, it must be a value between 1 and 12 inclusive for standard SKUs or between 1 and 3 inclusive for basic SKU.	int
semanticSearch	Sets options that control the availability of semantic search. This configuration is only possible for certain Azure Cognitive Search SKUs in certain locations.	'disabled' 'free' 'standard'

DataPlaneAuthOptions

Name	Description	Value
aadOrApiKey	Indicates that either the API key or an access token from Azure Active Directory can be used for authentication.	DataPlaneAadOrApiKeyAuthOption
apiKeyOnly	Indicates that only the API key needs to be used for authentication.	For Bicep, you can use the any() function.

DataPlaneAadOrApiKeyAuthOption

Name	Description	Value
aadAuthFailureMode	Describes what response the data plane API of a Search service would send for requests that failed authentication.	'http401WithBearerChallenge' 'http403'

EncryptionWithCmk

Name	Description	Value
enforcement	Describes how a search service should enforce having one or more non customer encrypted resources.	'Disabled' 'Enabled' 'Unspecified'

NetworkRuleSet

Name	Description	Value
bypass	Possible origins of inbound traffic that can bypass the rules defined in the 'ipRules' section.	'AzurePortal' 'None'
ipRules	A list of IP restriction rules that defines the inbound network(s) with allowing access to the search service endpoint. At the meantime, all other public IP networks are blocked by the firewall. These restriction rules are applied only when the 'publicNetworkAccess' of the search service is 'enabled'; otherwise, traffic over public interface is not allowed even with any public IP rules, and private endpoint connections would be the exclusive access method.	IpRule[]

IpRule

Name	Description	Value
value	Value corresponding to a single IPv4 address (eg., 123.1.2.3) or an IP range in CIDR format (eg., 123.1.2.3/24) to be allowed.	string

Sku

Name	Description	Value
name	The SKU of the search service. Valid values include: 'free': Shared service. 'basic': Dedicated service with up to 3 replicas. 'standard': Dedicated service with up to 12 partitions and 12 replicas. 'standard2': Similar to standard, but with more capacity per search unit. 'standard3': The largest Standard offering with up to 12 partitions and 12 replicas (or up to 3 partitions with more indexes if you also set the hostingMode property to 'highDensity'). 'storage_optimized_l1': Supports 1TB per partition, up to 12 partitions. 'storage_optimized_l2': Supports 2TB per partition, up to 12 partitions.'	'basic' 'free' 'standard' 'standard2' 'standard3' 'storage_optimized_l1' 'storage_optimized_l2'

Quickstart templates

The following quickstart templates deploy this resource type.

Template	Description
Web App with a SQL Database, Azure Cosmos DB, Azure Search Deploy to Azure	This template provisions a Web App, a SQL Database, Azure Cosmos DB, Azure Search and Application Insights.
Azure Cognitive Search service Deploy to Azure	This template creates an Azure Cognitive Search service
Azure Cognitive Search service with private endpoint Deploy to Azure	This template creates an Azure Cognitive Search service with a private endpoint.

Azure Policy built-in definitions for Azure Cognitive Search

Article • 03/08/2023 • 3 minutes to read

This page is an index of Azure Policy built-in policy definitions for Azure Cognitive Search. For additional Azure Policy built-ins for other services, see Azure Policy built-in definitions.

The name of each built-in policy definition links to the policy definition in th Azure portal. Use the link in the **Version** column to view the source on the Azure Policy GitHub repo.

Azure Cognitive Search

Name (Azure portal)	**Description**	**Effect(s)**	**Version** (GitHub)
Azure Cognitive Search service should use a SKU that supports private link	With supported SKUs of Azure Cognitive Search, Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The private link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to your Search service, data leakage risks are reduced. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should disable public network access	Disabling public network access improves security by ensuring that your Azure Cognitive Search service is not exposed on the public internet. Creating private endpoints can limit exposure of your Search service. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.	Audit, Deny, Disabled	1.0.0

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
Azure Cognitive Search services should have local authentication methods disabled	Disabling local authentication methods improves security by ensuring that Azure Cognitive Search services exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/azure-cognitive-search/rbac. Note that while the disable local authentication parameter is still in preview, the deny effect for this policy may result in limited Azure Cognitive Search portal functionality since some features of the Portal use the GA API which does not support the parameter.	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should use customer-managed keys to encrypt data at rest	Enabling encryption at rest using a customer-managed key on your Azure Cognitive Search services provides additional control over the key used to encrypt data at rest. This feature is often applicable to customers with special compliance requirements to manage data encryption keys using a key vault.	Audit, Deny, Disabled	1.0.0
Azure Cognitive Search services should use private link	Azure Private Link lets you connect your virtual network to Azure services without a public IP address at the source or destination. The Private Link platform handles the connectivity between the consumer and services over the Azure backbone network. By mapping private endpoints to Azure Cognitive Search, data leakage risks are reduced. Learn more about private links at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.	Audit, Disabled	1.0.0
Configure Azure Cognitive Search services to disable local authentication	Disable local authentication methods so that your Azure Cognitive Search services exclusively require Azure Active Directory identities for authentication. Learn more at: https://aka.ms/azure-cognitive-search/rbac.	Modify, Disabled	1.0.0
Configure Azure Cognitive Search services to disable public network access	Disable public network access for your Azure Cognitive Search service so that it is not accessible over the public internet. This can reduce data leakage risks. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.	Modify, Disabled	1.0.0

Name (Azure portal)	Description	Effect(s)	Version (GitHub)
Configure Azure Cognitive Search services with private endpoints	Private endpoints connect your virtual network to Azure services without a public IP address at the source or destination. By mapping private endpoints to your Azure Cognitive Search service, you can reduce data leakage risks. Learn more at: https://aka.ms/azure-cognitive-search/inbound-private-endpoints.	DeployIfNotExists, Disabled	1.0.0
Deploy Diagnostic Settings for Search Services to Event Hub	Deploys the diagnostic settings for Search Services to stream to a regional Event Hub when any Search Services which is missing this diagnostic settings is created or updated.	DeployIfNotExists, Disabled	2.0.0
Deploy Diagnostic Settings for Search Services to Log Analytics workspace	Deploys the diagnostic settings for Search Services to stream to a regional Log Analytics workspace when any Search Services which is missing this diagnostic settings is created or updated.	DeployIfNotExists, Disabled	1.0.0
Resource logs in Search services should be enabled	Audit enabling of resource logs. This enables you to recreate activity trails to use for investigation purposes; when a security incident occurs or when your network is compromised	AuditIfNotExists, Disabled	5.0.0

Next steps

- See the built-ins on the Azure Policy GitHub repo.
- Review the Azure Policy definition structure.
- Review Understanding policy effects.

Additional resources

Training

Learning path

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Azure Cognitive Search monitoring data reference

Article • 02/09/2023 • 6 minutes to read

This article provides a reference of log and metric data collected to analyze the performance and availability of Azure Cognitive Search. See Monitoring Azure Cognitive Search for an overview.

Metrics

This section lists the platform metrics collected for Azure Cognitive Search (Microsoft.Search/searchServices).

Metric ID	Unit	Description
DocumentsProcessedCount	Count	Total of the number of documents successfully processed in an indexing operation (either by an indexer or by pushing documents directly).
SearchLatency	Seconds	Average search latency for queries that execute on the search service.
SearchQueriesPerSecond	CountPerSecond	Average of the search queries per second (QPS) for the search service. It's common for queries to execute in milliseconds, so only queries that measure as seconds will appear in a metric like QPS. The minimum is the lowest value for search queries per second that was registered during that minute. The same applies to the maximum value. Average is the aggregate across the entire minute. For example, within one minute, you might have a pattern like this: one second of high load that is the maximum for SearchQueriesPerSecond, followed by 58 seconds of average load, and finally one second with only one query, which is the minimum.
SkillExecutionCount	Count	Total number of skill executions processed during an indexer operation.

Metric ID	Unit	Description
ThrottledSearchQueriesPercentage	Percent	Average percentage of the search queries that were throttled from the total number of search queries that executed during a one-minute interval.

For reference, see a list of all platform metrics supported in Azure Monitor.

Metric dimensions

Dimensions of a metric are name/value pairs that carry additional data to describe the metric value. Azure Cognitive Search has the following dimensions associated with its metrics that capture a count of documents or skills that were executed ("Document processed count" and "Skill execution invocation count").

Dimension Name	Description
DataSourceName	A named data source connection used during indexer execution. Valid values are one of the supported data source types.
Failed	Indicates whether the instance failed.
IndexerName	Name of an indexer.
IndexName	Name of an index.
SkillsetName	Name of a skillset used during indexer execution.
SkillName	Name of a skill within a skillset.
SkillType	The @odata.type of the skill.

For more information on what metric dimensions are, see Multi-dimensional metrics.

Resource logs

Resource logs are platform logs that provide insight into operations that were performed within an Azure resource. Resource logs are generated by the search service automatically, but are not collected by default. You must create a diagnostic setting to send resource logs to a Log Analytics workspace to use with Azure Monitor Logs, Azure Event Hubs to forward outside of Azure, or to Azure Storage for archiving.

This section identifies the type (or category) of resource logs you can collect for Azure Cognitive Search:

- Resource logs are grouped by type (or category). Azure Cognitive Search generates resource logs under the **Operations category**.

For reference, see a list of all resource logs category types supported in Azure Monitor.

Azure Monitor Logs tables

Azure Monitor Logs is a feature of Azure Monitor that collects and organizes log and performance data from monitored resources. If you configured a diagnostic setting for Log Analytics, you can query the Azure Monitor Logs tables for the resource logs generated by Azure Cognitive Search.

This section refers to all of the Azure Monitor Logs Kusto tables relevant to Azure Cognitive Search and available for query by Log Analytics and Metrics Explorer in the Azure portal.

Table	Description
AzureActivity	Entries from the Azure Activity log that provide insight into control plane operations. Tasks invoked on the control plane, such as adding or removing replicas and partitions, will be represented through a "Get Admin Key" activity.
AzureDiagnostics	Logged query and indexing operations.
AzureMetrics	Metric data emitted by Azure Cognitive Search that measures health and performance.

For a reference of all Azure Monitor Logs / Log Analytics tables, see the Azure Monitor Log Table Reference.

Diagnostics tables

Azure Cognitive Search uses the **Azure Diagnostics** table to collect resource logs related to queries and indexing on your search service.

Queries against this table in Log Analytics can include the common properties, the search-specific properties, and the search-specific operations listed in the schema reference section.

For examples of Kusto queries useful for Azure Cognitive Search, see Monitoring Azure Cognitive Search and Analyze performance in Azure Cognitive Search.

Activity logs

The following table lists common operations related to Azure Cognitive Search that may be created in the Azure Activity log.

Operation	Description
Get Admin Key	Any operation that requires administrative rights will be logged as a "Get Admin Key" operation.
Get Query Key	Any read-only operation against the documents collection of an index.
Regenerate Admin Key	A request to regenerate either the primary or secondary admin API key.

Common entries include references to API keys - generic informational notifications like *Get Admin Key* and *Get Query keys*. These activities indicate requests that were made using the admin key (create or delete objects) or query key, but do not show the request itself. For information of this grain, you must configure resource logging.

Alternatively, you might gain some insight through change history. In Azure portal, select the activity to open the detail page and then select "Change history" for information about the underlying operation.

For more information on the schema of Activity Log entries, see Activity Log schema.

Schemas

The following schemas are in use by Azure Cognitive Search. If you are building queries or custom reports, the data structures that contain Azure Cognitive Search resource logs conform to the schema below.

For resource logs sent to blob storage, each blob has one root object called **records** containing an array of log objects. Each blob contains records for all the operations that took place during the same hour.

Resource log schema

All resource logs available through Azure Monitor share a common top-level schema. Azure Cognitive Search supplements with additional properties and operations that are unique to a search service.

The following example illustrates a resource log that includes common properties (TimeGenerated, Resource, Category, and so forth) and search-specific properties

(OperationName and OperationVersion).

Name	Type	Description and example
TimeGenerated	Datetime	Timestamp of the operation. For example: `2021-12-07T00:00:43.6872559Z`
Resource	String	Resource ID. For example: `/subscriptions/<your-subscription-id>/resourceGroups/<your-resource-group-name>/providers/Microsoft.Search/searchServices/<your-search-service-name>`
Category	String	"OperationLogs". This value is a constant. OperationLogs is the only category used for resource logs.
OperationName	String	The name of the operation (see the full list of operations below). An example is `Query.Search`
OperationVersion	String	The api-version used on the request. For example: `2020-06-30`
ResultType	String	"Success". Other possible values: Success or Failure
ResultSignature	Int	An HTTP result code. For example: `200`
DurationMS	Int	Duration of the operation in milliseconds.
Properties	Object	Object containing operation-specific data. See the following properties schema table.

Properties schema

The properties below are specific to Azure Cognitive Search.

Name	Type	Description and example
Description_s	String	The operation's endpoint. For example: `GET /indexes('content')/docs`
Documents_d	Int	Number of documents processed.
IndexName_s	String	Name of the index associated with the operation.
Query_s	String	The query parameters used in the request. For example: `?search=beach access&$count=true&api-version=2020-06-30`

OperationName values (logged operations)

The operations below can appear in a resource log.

OperationName	Description
DataSources.*	Applies to indexer data sources. Can be Create, Delete, Get, List.
DebugSessions.*	Applies to a debug session. Can be Create, Delete, Get, List, Start, and Status.
DebugSessions.DocumentStructure	An enriched document is loaded into a debug session.
DebugSessions.RetrieveIndexerExecutionHistoricalData	A request for indexer execution details.
DebugSessions.RetrieveProjectedIndexerExecutionHistoricalData	Execution history for enrichments projected to a knowledge store.
Indexers.*	Applies to an indexer. Can be Create, Delete, Get, List, and Status.
Indexes.*	Applies to a search index. Can be Create, Delete, Get, List.
indexes.Prototype	This is an index created by the Import Data wizard.
Indexing.Index	This operation is a call to Add, Update or Delete Documents.
Metadata.GetMetadata	A request for search service system data.
Query.Autocomplete	An autocomplete query against an index. See Query types and composition.
Query.Lookup	A lookup query against an index. See Query types and composition.
Query.Search	A full text search request against an index. See Query types and composition.
Query.Suggest	Type ahead query against an index. See Query types and composition.

OperationName	Description
ServiceStats	This operation is a routine call to Get Service Statistics, either called directly or implicitly to populate a portal overview page when it is loaded or refreshed.
Skillsets.*	Applies to a skillset. Can be Create, Delete, Get, List.

See also

- See Monitoring Azure Cognitive Search for concepts and instructions.
- See Monitoring Azure resources with Azure Monitor for details on monitoring Azure resources.

Additional resources

🕮 Documentation

Monitor indexer status and results - Azure Cognitive Search

Monitor the status, progress, and results of Azure Cognitive Search indexers in the Azure portal, using the REST API, or the .NET SDK.

Analyze performance - Azure Cognitive Search

TBD

Estimate capacity for query and index workloads - Azure Cognitive Search

Adjust partition and replica computer resources in Azure Cognitive Search, where each resource is priced in billable search units.

Performance tips - Azure Cognitive Search

Learn about tips and best practices for maximizing performance on a search service.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Service limits for tiers and skus - Azure Cognitive Search

Service limits used for capacity planning and maximum limits on requests and responses for Azure Cognitive Search.

Update Indexer (Azure Cognitive Search REST API)

Modify an Azure Cognitive Search indexer used for crawling external data sources for searchable content.

Availability and continuity - Azure Cognitive Search

learn how to make a search service highly available and resilient against period disruptions or even catastrophic failures.

Show 5 more

Training

Learning paths and modules

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Learning certificate

Microsoft Certified: Azure Administrator Associate - Certifications

Azure administrators implement, manage, and monitor an organization's Microsoft Azure environment, including virtual networks, storage, compute, identity, security, and governance.

Stopwords reference (Microsoft analyzers)

Article • 05/17/2022 • 17 minutes to read

When text is indexed into Azure Cognitive Search, it's processed by analyzers so it can be efficiently stored in a search index. During this lexical analysis process, language analyzers will remove stopwords specific to that language. Stopwords are non-essential words such as "the" or "an" that can be removed without compromising the lexical integrity of your content.

Stopword removal applies to all supported Lucene and Microsoft analyzers used in Azure Cognitive Search.

This article lists the stopwords used by the Microsoft analyzer for each language.

For the stopword list for Lucene analyzers, see the Apache Lucene source code on GitHub.

> **Tip**
>
> To view the output of any given analyzer, call the **Analyze Text REST API**. This API is often helpful for debugging unexpected search results.

Arabic (ar.microsoft)

هذه هذا ما لا مع الذى الذي عن التى التي إلى الى على إن أن ان من فى في
كانت غير منذ كما قبل هو وفى وفي ذلك و أو او كل أنه انه إنه كان لم بعد
أيضا دون له لكن حول ومن الذين وهو هى هي قد فيها ولا وقد حتى اى اي أي وكان
أنها انها وأن وإن وان بها إذا اذا فيه أمام امام بعض إلا ألا الا حيث ايضا
عليها وعليها ومنها وكانت به وبه منها بل ولم وهى وهي لها عليه نحو ثم إنها
وأي وآخر وعند وأما وإما واما وحين وعبر لدى ومثل ولن ليس يمكن هناك عندما
وفقط وإم وام والان والآن وبما وتحت وبأن ومنه ومما وال وتلك وإذ واذ واى وأى
اى اي أى أي تلك اذ ولكن اخر آخر عند إما أما اما علي حين عبر وعلى مثل لن
ذ د خ ح ج ث ت ب ا ام فقط الآن بما ولكن وما تكون تحت بأن منه وهذا مما ال
ة ؤ ء ئ ى آ إ أ ي و ه ن م ل ك ق ف غ ع ظ ط ك ض ص ش س ز ر

Bengali (bn.microsoft)

во бы не на что по для как от это из за только или их все его он но до же то так уже а б в г д е ё ж з и й к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я

Bulgarian (bg.microsoft)

вж до е и на от то у че

Catalan (ca.microsoft)

la el l les de d del dels i un una uns unes a als al en és es s se hi

Chinese Simplified (zh-Hans.microsoft)

?about $ 1 2 3 4 5 6 7 8 9 0 _ a b c d e f g h i j k l m n o p q r s t u v w x y z after all also an and another any are as at be because been before being between both but by came can come could did do each for from get got had has have he her here him himself his how if in into is it like make many me might more most much must my never now of on only or other our out over said same see should since some still such take than that the their them then there these they this those through to too under up very was way we well were what where which while who with would you your 的 在 了 是 有 为 这 我 也 就 他 与 等 以 着 而 从 并 还 已 但 你 之 更 又 得 她 它 很 其 该 那 各

Chinese Traditional (zh-Hant.microsoft)

?about $ 1 2 3 4 5 6 7 8 9 0 _ a b c d e f g h i j k l m n o p q r s t u v w x y z after all also an and another any are as at be because been before being between both but by came can come could did do each for from get got had has have he her here him himself his how if in into is it like make many me might more most much must my never now of on only or other our out over said same see should since some still such take than that the their them then there these they this those through to too under up very was way we well were what where which while who with would you your 的 一 不 在 人 有 是 為 以 於 上 他 而 後 之 來 及 了 因 下 可 到 由 這 與 也 此 但 並 個 其 已 無 小 我 們 起 最 再 今 去 好 只 又 或 很 亦 某 把 那 你 乃 它

Croatian (hr.microsoft)

`i` `u` `je` `se` `na` `za` `da` `su` `o` `od` `a` `s`

Czech (cs.microsoft)

`na` `se` `je` `že` `do` `to` `ve` `ale` `za` `si` `pro` `po` `by` `od` `už` `který` `bude` `jako` `tak` `jsou` `jsem` `jsme` `však` `podle` `až` `jen` `ze` `před` `také` `jeho` `má` `když` `byl` `co` `jak` `nebo` `při` `ještě` `aby` `než` `budou` `ani` `jaké` `další` `kteří` `není` `bylo` `mezi` `v` `a` `i` `ač` `ačkoli` `přece` `no` `ne` `ano` `která` `které` `kterou` `kterými` `budu` `budeme` `budete` `byli` `byly` `o` `ať` `Á` `á` `Ě` `ě` `É` `é` `Í` `í` `Ó` `ó` `Ú` `ú` `Ů` `ů` `Ď` `ď` `Ň` `ň` `Č` `č` `Ř` `Š` `š` `ř` `Ť` `ť` `Ž` `ž` `Ý` `ý` `a` `b` `c` `d` `e` `f` `g` `h` `i` `j` `k` `l` `m` `n` `o` `p` `q` `r` `s` `t` `u` `v` `w` `x` `y` `z` `A` `B` `C` `D` `E` `F` `G` `H` `I` `J` `K` `L` `M` `N` `O` `P` `Q` `R` `S` `T` `U` `V` `W` `X` `Y` `Z` `0` `1` `2` `3` `4` `5` `6` `7` `8` `9` `I` `II` `III` `IV` `V` `VI` `VII` `VIII` `IX` `X` `XI` `XII` `XIII` `XIV` `XVI` `XVII` `XX` `X` `I` `V` `L` `D` `M` `C` `LD` `CCC` `IV` `VII` `XXXII` `LXXI` `VIII` `DCII` `LXX` `DCII.` `III` `DCCLII` `CDIX` `XXVII` `LIV` `MMVI` `MCMLXXII` `LXX` `DCC` `LII` `DCCIV` `DCV` `LXXIV` `LXV` `IXX` `XVIII`

Danish (da.microsoft)

`af` `alting` `at` `begge` `countries` `de` `dem` `den` `dén` `denne` `der` `deres` `det` `dét` `dette` `dig` `din` `dine` `disse` `dit` `du` `eder` `eders` `en` `én` `enhver` `ens` `er` `et` `ét` `ethvert` `for` `fra` `ham` `han` `hans` `har` `hende` `hendes` `hin` `hinanden` `hun` `hverandre` `hvis` `hvo` `I` `ikke` `ingen` `ingenting` `jeg` `jer` `jeres` `man` `med` `men` `mig` `min` `mine` `mit` `nogen` `nogle` `og` `om` `os` `på` `sig` `sin` `sine` `sit` `som` `somme` `til` `være` `vi` `vor` `Vor` `vore` `vores`

Dutch (nl.microsoft)

`de` `van` `en` `het` `in` `een` `is` `zijn` `de` `met` `op` `te` `voor` `die` `door` `dat` `aan` `tot` `als` `of` `in` `hij` `werd` `het` `uit` `bij` `ook` `niet` `wordt` `worden` `was` `er` `naar` `om` `zich` `maar` `heeft` `dan` `over` `deze` `nog` `meer` `kan` `ze` `hebben` `hun` `onder` `een` `kunnen` `tussen` `tegen` `dit` `na` `hij` `andere` `al` `zij` `veel` `men` `geen` `werden` `wel` `waar` `zie` `vooral` `weer` `deel` `je` `wat` `nu` `ten` `alle` `op` `van` `had` `waren` `maar` `moet` `zo` `zeer` `hem` `bij` `ook`

English (en.microsoft)

`is` `and` `in` `it` `of` `the` `to` `that` `this` `these` `those` `is` `was` `for` `on` `be` `with` `as` `by` `at` `have` `are` `this` `not` `but` `had` `from` `or` `I` `my` `me` `mine` `myself` `you` `your` `yours` `yourself` `he` `him` `his` `himself` `she` `her` `hers` `herself` `it` `its` `itself` `we` `our` `ours` `ourselves`

they them their theirs themselves A B C D E F G H J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

Finnish (fi.microsoft)

ai ainoa ainoaa ainoaan ainoaksi ainoalla ainoalle ainoalta ainoan ainoassa ainoasta ali alitse alla alle alta edellä edelle edeltä ehkä ei enemmän eniten ennen entä entäs erääkseen erääksi eräällä erälle eräältä erään eräässä eräästä eräs eräskin erästä että hän häneen häneksi hänellä hänelle häneltä hänen hänessä hänestä harva harvaa harvaan harvaksi harvalla harvalle harvalta harvan harvassa harvasta harvat harvoihin harvoiksi harvoilla harvoille harvoilta harvoissa harvoista harvoja harvojen he heidän heihin heiksi heillä heille heiltä heissä heistä hiukan huolimatta ilman itse itseäni itseeni itseensä itsekseni itselläni itselleni itseni itsessäni itsestäni ja jälkeen johon johonkin joiden joidenkin joihin joihinkin joiksikin joilla joillakin joille joillekin joilta joiltakin joissa joissakin joista joistakin joita joka jokainen jokaiseksi jokaisella jokaiselle jokaiselta jokaisen jokaisessa jokaisesta jokaista jokin joko joksikin jokunen jolaiseen jolla jollainen jollaiseen jollaiseksi jollaisella jollaiselle jollaiselta jollaisen jollaisessa jollaisesta jollaiset jollaisia jollaisiin jollaisiksi jollaisilla jollaisille jollaisilta jollaisissa jollaisista jollaista jollaisten jollakin jolle jollekin jolta jonka jonkin jonkinlainen jonkinlaiseen jonkinlaiseksi jonkinlaisella jonkinlaiselle jonkinlaiselta jonkinlaisen jonkinlaisessa jonkinlaisesta jonkinlaiset jonkinlaisia jonkinlaisiin jonkinlaisiksi jonkinlaisilla jonkinlaisille jonkinlaisilta jonkinlaisissa jonkinlaisista jonkinlaista jonkinlaisten jos jossa jossakin josta jostakin jota jotakin jotka jotkin jotta kaikki kanssa kauas kaukana kautta keiden keitä kenen kera keskellä keskelle ketä ketkä kohti koska kuhunkin kuin kuinka kuitenkin kuka kukin kullakin kullekin kuluessa kuluttua kummallakin kummallekin kummaltakin kummankin kummassakin kummastakin kummatkin kumpaakin kumpaankin kumpanakin kumpiakin kumpienkin kumpikin kun kunkin kussakin kutakin kyllä kylläkin lähellä lähelle läpi lisäksi lukuunottamatta luokse luona luota me meidän meihin meiksi meillä meille meiltä meissä meistä meitä mikä miksei miksi milloin minä minua minuksi minulla minulle minulta minun minussa minusta minuun mistä miten mitkä molemmat molemmiksi molemmilla molemmille molemmissa molemmista molempia molempien molempiin moneen moneksi monella monelle monelta monen monenlainen monenlaiseen monenlaiseksi monenlaisella monenlaiselle monenlaiselta monenlaisen monenlaisessa

monenlaisesta monenlaiset monenlaisia monenlaisiin monenlaisiksi monenlaisilla monenlaisille monenlaisilta monenlaisissa monenlaisista monenlaista monenlaisten monessa monesta moni monia monien moniin moniksi monilla monille monilta monissa monista monta montaa muihin muiksi muilla muille muilta muissa muista muita mukana mukanaan mutta muu muuan muuhun muuksi muulla muulle muulta muun muussa muusta muut muuta muutaista muutama muutamaa muutamaan muutamaksi muutamalla muutamalle muutamalta muutaman muutamassa muutamasta muutamat muutamia muutamien muutamiin muutamiksi muutamilla muutamille muutamilta muutamissa muutamista myös myöten näet näiden näihin näiksi näille nämä ne niiden niihin niiksi niille nimittäin no noiden noihin noiksi noille nuo ohhoh ohi ohitse paikkeilla päin paitsi paljon pitkin pois poispäin sama samaa samaan samaksi samalla samalle samalta saman samassa samasta samat samoihin samoiksi samoilla samoille samoilta samoissa samoista samoja samojen se sellainen sellaiseen sellaiseksi sellaisella sellaiselle sellaiselta sellaisen sellaisessa sellaisesta sellaiset sellaisia sellaisiin sellaisiksi sellaisilla sellaisille sellaisilta sellaisissa sellaisista sellaista sellaisten sen siihen siinä siis siitä siksi sillä sille siltä silti sinä sinua sinuksi sinulla sinulle sinulta sinun sinussa sinusta sinuun sitä taakse taas tähän tai takaa takana täksi tällainen tällaiseen tällaiseksi tällaisella tällaiselle tällaiselta tällaisen tällaisessa tällaisesta tällaiset tällaisia tällaisiin tällaisiksi tällaisilla tällaisille tällaisilta tällaisissa tällaisista tällaista tällaisten tälle tämä tämän te teidän toki tosin tuo tuohon tuoksi tuolle usea useaa useaan useaksi usealla usealle usealta usean useassa useasta useat useiden useiksi useilla useille useilta useisiin useissa useista useita vähän vähemmän vähiten vai vaikka vailla varten vasten vastoin vielä vieläkin voi yli yllä ylle yltä ympäri ympärillä ympärille

French (fr.microsoft)

ces cet cette de des du es et la le les on un une

German (de.microsoft)

aber alle aller alles als am an auch auf aus bei bis dann das daß dein dem den der deren des dessen die diese dieser dieses du durch ein eine einem einen einer eines einige einigem einigen einiger einiges er es etliche etlichem etlichen etlicher etliches euer eurer für gegen habe haben hat hatte ich ihr

`ihre` `im` `immer` `in` `ist` `jede` `jedem` `jeden` `jeder` `jedes` `jene` `jener` `jenes` `kann` `kein` `keine` `keinem` `keinen` `können` `man` `manche` `manchem` `manchen` `mancher` `manches` `mehr` `mein` `mit` `nach` `nicht` `noch` `nur` `oder` `schon` `sei` `sein` `seine` `seiner` `sich` `sie` `sind` `so` `soll` `über` `um` `und` `unser` `unter` `vom` `von` `vor` `war` `welche` `welcher` `welches` `wenn` `werden` `wessen` `wie` `wieder` `wir` `wird` `worden` `wurde` `zu` `zum` `zur` `zwei` `zwischen` `a` `ä` `b` `c` `d` `e` `f` `g` `h` `i` `j` `k` `l` `m` `n` `o` `ö` `p` `q` `r` `s` `t` `u` `ü` `v` `w` `x` `y` `z` `ß` `A` `Ä` `B` `C` `D` `E` `F` `G` `H` `I` `J` `K` `L` `M` `N` `O` `Ö` `P` `Q` `R` `S` `T` `U` `Ü` `V` `W` `X` `Y` `Z` `é`

Greek (el.microsoft)

`ο` `η` `το` `στο` `οι` `τα` `του` `εμάς` `εσάς` `εσένα` `εμένα` `σου` `αι` `ημών` `μένα` `σένα` `ων` `όντας` `εμέ` `σας` `μας` `σεις` `τοις` `τω` `υμάς` `υμείς` `υμών` `εσέ` `μείς` `μού` `σού` `τού` `τής` `τόν` `τήν` `μου` `τό` `μάς` `σάς` `τούς` `τά` `δικό` `δικός` `δικές` `δικών` `δική` `δικής` `δικήν` `δικιά` `δικιάν` `δικά` `δικιάς` `δικιές` `δικοί` `δικού` `δικούς` `δικόν` `της` `των` `τον` `την` `το` `τους` `τις` `τα` `τη` `ένας` `μια` `ένα` `ενός` `μιας` `με` `σε` `αν` `εάν` `να` `δια` `εκ` `εξ` `επί` `προ` `υπέρ` `από` `προς` `και` `ούτε` `μήτε` `ουδέ` `μηδέ` `ή` `είτε` `μα` `αλλά` `παρά` `όμως` `ωστόσο` `ενώ` `μολονότι` `μόνο` `πώς` `που` `πριν` `οτί` `λοιπόν` `ώστε` `άρα` `επομένως` `όταν` `σαν` `καθώς` `αφού` `αφότου` `πριν` `μόλις` `άμα` `προτού` `ως` `ώσπου` `όσο` `ωσότου` `όποτε` `κάθε` `γιατί` `επειδή` `ίσως` `παρά` `θα` `ας` `τι` `αντί` `μετά` `κατά` `από` `προς` `αλλά` `για` `τες` `κι` `σ'` `απ'` `γι'` `συ` `μ'` `κατ'` `ουτ'` `στ'` `παρ'` `τόσο` `τι` `όσο` `ό,τι` `θα` `όπου` `δε` `εάν` `εγώ` `εσύ` `αυτός` `αυτή` `αυτό` `εμείς` `εσείς` `αυτοί` `αυτές` `αυτά` `αυτών` `αυτούς` `αυτές` `πιο` `εδώ` `εκεί` `έτσι` `στα` `στων` `πλέον` `ακόμα` `τώρα` `τότε` `όταν` `ούτως` `άλλως` `αλλιώς` `συνεπώς` `εξής` `τούδε` `εφεξής` `όθεν` `οσοδήποτε` `εντούτοις` `μολαταύτα` `έστω` `παρόλο` `πια` `καθόλου` `καν` `χωρίς` `οποτεδήποτε` `πράγματι` `όντως` `άραγε` `μολοντούτο` `απολύτως` `παρομοίως` `σάμπως` `άκρως` `υπό` `ειδεμή` `δηλάδή` `ήτοι` `μέσω` `περί` `περίπου` `α` `Α` `β` `Β` `γ` `Γ` `δ` `Δ` `ε` `Ε` `ζ` `Ζ` `η` `Η` `θ` `Θ` `ι` `Ι` `κ` `Κ` `λ` `Λ` `μ` `Μ` `ν` `Ν` `ξ` `Ξ` `ο` `Ο` `π` `Π` `ρ` `Ρ` `σ` `ς` `Σ` `τ` `Τ` `υ` `Υ` `φ` `Φ` `χ` `Χ` `ψ` `Ψ` `ω` `Ω` `ϊ` `ΐ` `Ϊ` `ϋ` `ΰ` `Ϋ`

Gujarati (gu.microsoft)

`માં` `તે` `એ` `આ` `છે` `અને` `નો` `ની` `નું`

Hebrew (he.microsoft)

`את` `אל` `של`

Hindi (hi.microsoft)

है हैं में और का की के वह यह

Hungarian (hu.microsoft)

a az és hogy nem is de szerint már csak meg még ez volt mint azt vagy pedig aztán ha akkor izé szintén ki után kell majd van aki azonban lesz mert illetve amely akkor lehet nagyon miatt ami sem a á b c cs d dz dzs e é f g gy h i í j k l ly m n ny o ó ö ő p q r s sz t ty u ú ü ű v w x y z zs A Á B C Cs D Dz Dzs E É F G Gy H I Í J K L Ly M N Ny O Ó Ö Ő P Q R S Sz T Ty U Ú Ü Ű V W X Y Z Zs

Icelandic (is.microsoft)

að í á það eru er og

Indonesian (Bahasa) (id.microsoft)

ah di dong ialah ini itu juga ke sih

Italian (it.microsoft)

a agli ai al all' alla alle allo d' degli dei del dell' della delle dello di e è gli i il in l' la le lo negli nei nel nell' nella nelle nello un' un una uno

Japanese (ja.microsoft)

a and in is it of the to の を に は が と で

Kannada (kn.microsoft)

ಅ ಈ ಮತ್ತು ಮತ್ತೆ ಇರುತ್ತಾನೆ ಇರುತ್ತಾಳೆ ಇರುತ್ತದೆ ಇದು ಇದನ್ನು ಇದರ ಅದು ಅದನ್ನು ಅದರ

Latvian (lv.microsoft)

no un uz ir

Lithuanian (lt.microsoft)

`ir` `arba` `yra` `jis` `ji`

Malayalam (ml.microsoft)

`ഒരു` `ആണ്` `ആൺ` `ഇത്` `അത്` `ഈ` `ആ`

Malay (Latin) (ms.microsoft)

`adalah` `atau` `dalam` `dan` `di` `ia` `ialah` `ini` `itu` `juga` `lah`

Marathi (mr.microsoft)

`होते` `त्या` `होता` `होती` `होतो` `होतं` `हा` `हे` `ही` `ह्या` `हें` `हीं` `ती` `ते` `ती` `त्या` `तें` `तीं` `त्यां`

Norwegian (nb.microsoft)

`av` `og` `en` `ei` `et` `til` `i` `er` `den` `det`

Polish (pl.microsoft)

`a` `do` `i` `jest` `na` `o` `ta` `te` `to` `w` `we` `z` `za` `się` `że` `od` `przez` `po` `dla` `jak` `tym` `ale` `ma` `co` `czy` `oraz` `może` `tego` `tylko` `jednak` `jego` `już` `lub` `ich` `ze` `tak` `być` `być` `jestem` `jesteś` `jest` `jesteśmy` `jesteście` `są` `będę` `będziesz` `będzie` `będziemy` `będziecie` `będą` `byłem` `byłam` `byłeś` `byłaś` `był` `była` `było` `byliśmy` `byłyśmy` `byliście` `byłyście` `byli` `były` `byłbym` `byłabym` `byłbyś` `byłabyś` `byłby` `byłaby` `byłoby` `bylibyśmy` `byłybyśmy` `bylibyście` `byłybyście` `byliby` `byłyby` `bądź` `bądźmy` `bądźcie` `będąc` `byłże` `byłaże` `byłoże` `byliże` `byłyże` `które` `która` `też` `także` `który` `tej` `przed` `można` `jej` `przy` `ten` `pod` `jeszcze` `gdy` `jako` `by` `bardzo` `bo` `jeśli` `tych` `więc` `bez` `również` `nawet` `temu` `tm` `tymi` `ja` `mnie` `mię` `mi` `mną` `ty` `ciebie` `cię` `tobie` `ci` `tobą` `on` `go` `mu` `jemu` `niego` `nim` `niemu` `my` `nas` `nam` `nami` `wy` `was` `wam` `wami` `oni` `im` `nich` `nimi` `mój` `mojego` `mego` `mojemu` `memu` `moim` `mym` `moja` `mojej` `mej` `moją` `mą` `moi` `moje` `moich` `mych` `me` `moimi` `mymi` `twój` `twojego` `twego` `twojemu` `twemu` `twoim` `twym` `twoja` `twa` `twojej` `twej` `twoją` `twą` `twoi` `twoje` `twe` `twoich` `twych` `twoimi` `twymi` `nasz` `naszego` `naszemu` `naszym` `nasza` `naszej` `naszą` `nasi` `nasze` `naszych` `naszymi` `wasz` `waszego` `waszemu` `waszym` `wasza` `waszej` `waszą` `wasi` `wasze` `waszych` `waszymi` `one` `ą` `b` `c` `ć` `d` `e` `ę` `f` `g` `h` `j` `k` `l` `ł` `m` `n` `ń` `ó` `p` `q` `r` `s`

`ś` `t` `u` `v` `x` `y` `ź` `ż` `A` `Ą` `B` `C` `Ć` `D` `E` `Ę` `F` `G` `H` `I` `J` `K` `L` `Ł` `M` `N` `Ń` `O` `Ó` `P` `Q` `R` `S` `Ś` `T` `U` `V` `W` `X` `Y` `Z` `Ź` `Ż`

Portuguese (Brazil) (pt-Br.microsoft)

`à` `às` `é` `a` `ao` `aos` `as` `da` `das` `de` `do` `dos` `e` `em` `na` `nas` `no` `nos` `o` `os` `para` `um` `uma` `umas` `uns`

Portuguese (Portugal) (pt-Pt.microsoft)

`à` `às` `é` `a` `ao` `aos` `as` `da` `das` `de` `do` `dos` `e` `em` `na` `nas` `no` `nos` `o` `os` `para` `um` `uma` `umas` `uns`

Punjabi (pa.microsoft)

`ਹੈ` `ਹਨ` `ਦਾ` `ਦੇ` `ਦੀ` `ਦੀਆਂ` `ਵਿਚ` `ਅਤੇ` `ਇਹ` `ਉਹ`

Romanian (ro.microsoft)

`a` `ai` `al` `ale` `alor` `de` `din` `este` `în` `într` `la` `o` `pe` `și` `un` `unei` `unor` `unui`

Russian (ru.microsoft)

`во` `бы` `не` `на` `что` `по` `для` `как` `от` `это` `из` `за` `только` `или` `их` `все` `его` `он` `но` `до` `же` `то` `так` `уже` `а` `б` `в` `г` `д` `е` `ё` `ж` `з` `и` `й` `к` `л` `м` `н` `о` `п` `р` `с` `т` `у` `ф` `х` `ц` `ч` `ш` `щ` `ъ` `ы` `ь` `э` `ю` `я`

Serbian (Cyrillic) (sr-cyrillic.microsoft)

`и` `је` `у` `да` `се` `на` `су` `за` `од` `са` `а` `из` `о`

Serbian (Latin) (sr-latin.microsoft)

`i` `je` `u` `se` `su` `a`

Slovak (sk.microsoft)

z od na k o na a k v vo na je ono

Slovenian (sl.microsoft)

in k h v je ono onega onemu onem onim

Spanish (es.microsoft)

a al de del e el en es la las lo los un una unas unos y

Swedish (sv.microsoft)

av och en ett till i är den det

Tamil (ta.microsoft)

அது இது ஒரு அல்லது இந்த அந்த

Telugu (te.microsoft)

అతను ఆయన వారు వాండ్ళు వాళ్ళు ఆమె ఆనిడ అది అవి ఇతను ఈయవ వీరు వీండ్ళు ఈమె ఈనిడ

Thai (th.microsoft)

นะ ครับ ค่ะ ละ ฮะ ฮิ จ๊ะ วะ มั๊ย สิ เออ ฮึ ซิ นะจ๊ะ นะค่ะ เถอะ เถอะนะ เถอะน่า หรอก อุ๊ย อ่า ซะ หนิ หนะ หน่า นิ แหละ จ๊อกๆ ติ๋งๆ เหรอะ บรึม อื๋ย แหนะ เฮ้ย โว้ย โอย เจี๊ยก หรือก บ๊ะ แน่ะ โอ แฮ้ เหม่ เอ๊ะ แฮ แฮะ

Turkish (tr.microsoft)

ama ancak bazı bir çok da daha de değil diye en gibi göre hem her için ile ise kadar ki sadece üzere ve veya ya a b c ç d e f g ğ h ı i j k l m n o ö p r s ş t u ü v y z A B C Ç D E F G Ğ H I İ J K L M N O Ö P R S Ş T U Ü V Y Z ben sen o biz siz onlar bu şu hangi kendi bazı çok kim beni bana bende benden benim benimle bensiz seni sana sende senden seninle sensiz onu ona onda ondan onunla onsuz bizi bize bizde bizden bizimle bizsiz sizi size sizde

`sizden` `sizinle` `sizsiz` `onları` `onlara` `onlarda` `onlardan` `onların` `onlarla` `onlarsız` `bazısı` `bazısını` `bazısına` `bazısında` `bazısından` `bazısının` `bazısıyla` `bazılarımız` `bazılarınız` `bazılarına` `bazılarını` `bazılarından` `bazılarıyla` `bazılarımızı` `bazılarınızı` `bazılarımıza` `bazılarınıza` `bazılarımızda` `bazılarınızda` `bazılarımızdan` `bazılarınızdan` `bazılarımızla` `bazılarınızla` `kimileri` `kimilerini` `kimilerine` `kimilerinde` `kimilerinden` `kimileriyle` `kimilerimiz` `kimilerimizi` `kimilerimize` `kimilerimizde` `kimilerimizden` `kimilerimizin` `kimilerimizle` `kimilerimizsiz` `kimileriniz` `kimilerinizi` `kimilerinize` `kimilerinizde` `kimilerinizden` `kimilerinizin` `kimilerinizle` `kimilerinizsiz` `kendim` `kendimi` `kendime` `kendimde` `kendimden` `kendimin` `kendimle` `kendimsiz` `kendin` `kendini` `kendine` `kendinde` `kendinden` `kendinin` `kendinle` `kendinsiz` `kendisi` `kendisini` `kendisine` `kendisinde` `kendisinden` `kendisiyle` `kendileri` `kendilerini` `kendilerine` `kendilerinde` `kendilerinden` `kendileriyle` `kaçımız` `kaçımızı` `kaçımıza` `kaçımızda` `kaçımızdan` `kaçımızla` `hangimiz` `hangimizi` `hangimize` `hangimizde` `hangimizden` `hangimizle` `hangimizsiz` `bunu` `buna` `bunda` `bundan` `bunun` `bununla` `bunsuz` `bunlar` `bunları` `bunlara` `bunlarda` `bunlardan` `bunların` `bunlarla` `bunlarsız` `şunu` `şuna` `şunda` `şundan` `şunun` `şununla` `şunsuz` `şunlar` `şunları` `şunlara` `şunlarda` `şunlardan` `şunların` `şunlarla` `şunlarsız`

Ukrainian (uk.microsoft)

`і` `й` `до` `в` `у` `є` `з` `із`

Urdu (ur.microsoft)

`ان` `اس` `وہ` `یہ` `کے` `کی` `کا` `اور` `میں` `ہیں` `ہے`

See also

- Tutorial: Create a custom analyzer for phone numbers
- Add language analyzers to string fields
- Add custom analyzers to string fields
- Full text search in Azure Cognitive Search
- Analyzers for text processing in Azure Cognitive Search

Additional resources

🕮 Documentation

Create Synonym Map (Azure Cognitive Search REST API)

A synonym map to expand or rewrite a search query can be created using REST API in Azure Cognitive Search.

Search Documents (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Search Documents REST API for Azure Cognitive Search.

Tutorial: create a custom analyzer - Azure Cognitive Search

Learn how to build a custom analyzer to improve the quality of search results in Azure Cognitive Search.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

Add scoring profiles - Azure Cognitive Search

Boost search relevance scores for Azure Cognitive Search results by adding scoring profiles to a search index.

Create or Update Index (2021-04-30-Preview) - Azure Cognitive Search

Preview version of the Create or Update Index REST API for Azure Cognitive Search.

Map fields in indexers - Azure Cognitive Search

Configure field mappings in an indexer to account for differences in field names and data representations.

Relevance and scoring - Azure Cognitive Search

Explains the concepts of relevance and scoring in Azure Cognitive Search, and what a developer can do to customize the scoring result.

Show 5 more

OData language overview for `$filter`, `$orderby`, and `$select` in Azure Cognitive Search

Article • 07/19/2022 • 7 minutes to read

This article provides an overview of the OData expression language used in $filter, $order-by, and $select expressions in Azure Cognitive Search. The language is presented "bottom-up" starting with the most basic elements. The OData expressions that you can construct in a query request range from simple to highly complex, but they all share common elements. Shared elements include:

- **Field paths**, which refer to specific fields of your index.
- **Constants**, which are literal values of a certain data type.

Once you understand these common concepts, you can continue with the top-level syntax for each expression:

- **$filter** expressions are evaluated during query parsing, constraining search to specific fields or adding match criteria used during index scans.
- **$orderby** expressions are applied as a post-processing step over a result set to sort the documents that are returned.
- **$select** expressions determine which document fields are included in the result set.

The syntax of these expressions is distinct from the simple or full query syntax used in the **search** parameter, although there's some overlap in the syntax for referencing fields.

> **Note**
>
> Terminology in Azure Cognitive Search differs from the **OData standard** in a few ways. What we call a **field** in Azure Cognitive Search is called a **property** in OData, and similarly for **field path** versus **property path**. An **index** containing **documents** in Azure Cognitive Search is referred to more generally in OData as an **entity set** containing **entities**. The Azure Cognitive Search terminology is used throughout this reference.

Field paths

The following EBNF (Extended Backus-Naur Form) defines the grammar of field paths.

```
field_path ::= identifier('/'identifier)*

identifier ::= [a-zA-Z_][a-zA-Z_0-9]*
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

A field path is composed of one or more **identifiers** separated by slashes. Each identifier is a sequence of characters that must start with an ASCII letter or underscore, and contain only ASCII letters, digits, or underscores. The letters can be upper- or lower-case.

An identifier can refer either to the name of a field, or to a **range variable** in the context of a collection expression (`any` or `all`) in a filter. A range variable is like a loop variable that represents the current element of the collection. For complex collections, that variable represents an object, which is why you can use field paths to refer to subfields of the variable. This is analogous to dot notation in many programming languages.

Examples of field paths are shown in the following table:

Field path	Description
`HotelName`	Refers to a top-level field of the index
`Address/City`	Refers to the `City` subfield of a complex field in the index; `Address` is of type `Edm.ComplexType` in this example
`Rooms/Type`	Refers to the `Type` subfield of a complex collection field in the index; `Rooms` is of type `Collection(Edm.ComplexType)` in this example
`Stores/Address/Country`	Refers to the `Country` subfield of the `Address` subfield of a complex collection field in the index; `Stores` is of type `Collection(Edm.ComplexType)` and `Address` is of type `Edm.ComplexType` in this example
`room/Type`	Refers to the `Type` subfield of the `room` range variable, for example in the filter expression `Rooms/any(room: room/Type eq 'deluxe')`

Field path	Description
`store/Address/Country`	Refers to the `Country` subfield of the `Address` subfield of the `store` range variable, for example in the filter expression `Stores/any(store: store/Address/Country eq 'Canada')`

The meaning of a field path differs depending on the context. In filters, a field path refers to the value of a *single instance* of a field in the current document. In other contexts, such as **$orderby**, **$select**, or in fielded search in the full Lucene syntax, a field path refers to the field itself. This difference has some consequences for how you use field paths in filters.

Consider the field path `Address/City`. In a filter, this refers to a single city for the current document, like "San Francisco". In contrast, `Rooms/Type` refers to the `Type` subfield for many rooms (like "standard" for the first room, "deluxe" for the second room, and so on). Since `Rooms/Type` doesn't refer to a *single instance* of the subfield `Type`, it can't be used directly in a filter. Instead, to filter on room type, you would use a lambda expression with a range variable, like this:

OData

```
Rooms/any(room: room/Type eq 'deluxe')
```

In this example, the range variable `room` appears in the `room/Type` field path. That way, `room/Type` refers to the type of the current room in the current document. This is a single instance of the `Type` subfield, so it can be used directly in the filter.

Using field paths

Field paths are used in many parameters of the Azure Cognitive Search REST APIs. The following table lists all the places where they can be used, plus any restrictions on their usage:

API	Parameter name	Restrictions
Create or Update Index	`suggesters/sourceFields`	None
Create or Update Index	`scoringProfiles/text/weights`	Can only refer to **searchable** fields
Create or Update Index	`scoringProfiles/functions/fieldName`	Can only refer to **filterable** fields

API	Parameter name	Restrictions
Search	`search` when `queryType` is `full`	Can only refer to **searchable** fields
Search	`facet`	Can only refer to **facetable** fields
Search	`highlight`	Can only refer to **searchable** fields
Search	`searchFields`	Can only refer to **searchable** fields
Suggest and Autocomplete	`searchFields`	Can only refer to fields that are part of a suggester
Search, Suggest, and Autocomplete	`$filter`	Can only refer to **filterable** fields
Search and Suggest	`$orderby`	Can only refer to **sortable** fields
Search, Suggest, and Lookup	`$select`	Can only refer to **retrievable** fields

Constants

Constants in OData are literal values of a given Entity Data Model (EDM) type. See Supported data types for a list of supported types in Azure Cognitive Search. Constants of collection types aren't supported.

The following table shows examples of constants for each of the data types supported by Azure Cognitive Search:

Data type	Example constants
`Edm.Boolean`	`true`, `false`
`Edm.DateTimeOffset`	`2019-05-06T12:30:05.451Z`
`Edm.Double`	`3.14159`, `-1.2e7`, `NaN`, `INF`, `-INF`
`Edm.GeographyPoint`	`geography'POINT(-122.131577 47.678581)'`
`Edm.GeographyPolygon`	`geography'POLYGON((-122.031577 47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))'`

Data type	Example constants
`Edm.Int32`	`123`, `-456`
`Edm.Int64`	`283032927235`
`Edm.String`	`'hello'`

Escaping special characters in string constants

String constants in OData are delimited by single quotes. If you need to construct a query with a string constant that might itself contain single quotes, you can escape the embedded quotes by doubling them.

For example, a phrase with an unformatted apostrophe like "Alice's car" would be represented in OData as the string constant `'Alice''s car'`.

> **Important**
>
> When constructing filters programmatically, it's important to remember to escape string constants that come from user input. This is to mitigate the possibility of **injection attacks**, especially when using filters to implement **security trimming**.

Constants syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar for most of the constants shown in the above table. The grammar for geo-spatial types can be found in OData geo-spatial functions in Azure Cognitive Search.

```
constant ::=
    string_literal
    | date_time_offset_literal
    | integer_literal
    | float_literal
    | boolean_literal
    | 'null'

string_literal ::= "'"([^'] | "''")*"'"

date_time_offset_literal ::= date_part'T'time_part time_zone

date_part ::= year'-'month'-'day
```

```
time_part ::= hour':'minute(':'second('.'fractional_seconds)?)?

zero_to_fifty_nine ::= [0-5]digit

digit ::= [0-9]

year ::= digit digit digit digit

month ::= '0'[1-9] | '1'[0-2]

day ::= '0'[1-9] | [1-2]digit | '3'[0-1]

hour ::= [0-1]digit | '2'[0-3]

minute ::= zero_to_fifty_nine

second ::= zero_to_fifty_nine

fractional_seconds ::= integer_literal

time_zone ::= 'Z' | sign hour':'minute

sign ::= '+' | '-'

/* In practice integer literals are limited in length to the precision of
the corresponding EDM data type. */
integer_literal ::= digit+

float_literal ::=
    sign? whole_part fractional_part? exponent?
    | 'NaN'
    | '-INF'
    | 'INF'

whole_part ::= integer_literal

fractional_part ::= '.'integer_literal

exponent ::= 'e' sign? integer_literal

boolean_literal ::= 'true' | 'false'
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> ⓘ **Note**
>
> See OData expression syntax reference for Azure Cognitive Search for the complete EBNF.

Building expressions from field paths and constants

Field paths and constants are the most basic part of an OData expression, but they're already full expressions themselves. In fact, the **$select** parameter in Azure Cognitive Search is nothing but a comma-separated list of field paths, and **$orderby** isn't much more complicated than **$select**. If you happen to have a field of type `Edm.Boolean` in your index, you can even write a filter that is nothing but the path of that field. The constants `true` and `false` are likewise valid filters.

However, most of the time you'll need more complex expressions that refer to more than one field and constant. These expressions are built in different ways depending on the parameter.

The following EBNF (Extended Backus-Naur Form) defines the grammar for the **$filter**, **$orderby**, and **$select** parameters. These are built up from simpler expressions that refer to field paths and constants:

```
filter_expression ::= boolean_expression

order_by_expression ::= order_by_clause(',' order_by_clause)*

select_expression ::= '*' | field_path(',' field_path)*
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

Next steps

The **$orderby** and **$select** parameters are both comma-separated lists of simpler expressions. The **$filter** parameter is a Boolean expression that is composed of simpler subexpressions. These subexpressions are combined using logical operators such as

and, or, and not, comparison operators such as eq, lt, gt, and so on, and collection operators such as any and all.

The **$filter**, **$orderby**, and **$select** parameters are explored in more detail in the following articles:

- OData $filter syntax in Azure Cognitive Search
- OData $orderby syntax in Azure Cognitive Search
- OData $select syntax in Azure Cognitive Search

OData $filter syntax in Azure Cognitive Search

Article • 07/19/2022 • 7 minutes to read

In Azure Cognitive Search, the **$filter** parameter specifies inclusion or exclusion criteria for returning matches in search results. This article describes the OData syntax of **$filter** and provides examples.

Field path construction and constants are described in the OData language overview in Azure Cognitive Search. For more information about filter scenarios, see Filters in Azure Cognitive Search.

Syntax

A filter in the OData language is a Boolean expression, which in turn can be one of several types of expression, as shown by the following EBNF (Extended Backus-Naur Form):

```
boolean_expression ::=
    collection_filter_expression
    | logical_expression
    | comparison_expression
    | boolean_literal
    | boolean_function_call
    | '(' boolean_expression ')'
    | variable

/* This can be a range variable in the case of a lambda, or a field path. */
variable ::= identifier | field_path
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

The types of Boolean expressions include:

- Collection filter expressions using `any` or `all`. These apply filter criteria to collection fields. For more information, see OData collection operators in Azure Cognitive Search.
- Logical expressions that combine other Boolean expressions using the operators `and`, `or`, and `not`. For more information, see OData logical operators in Azure Cognitive Search.
- Comparison expressions, which compare fields or range variables to constant values using the operators `eq`, `ne`, `gt`, `lt`, `ge`, and `le`. For more information, see OData comparison operators in Azure Cognitive Search. Comparison expressions are also used to compare distances between geo-spatial coordinates using the `geo.distance` function. For more information, see OData geo-spatial functions in Azure Cognitive Search.
- The Boolean literals `true` and `false`. These constants can be useful sometimes when programmatically generating filters, but otherwise don't tend to be used in practice.
- Calls to Boolean functions, including:
  - `geo.intersects`, which tests whether a given point is within a given polygon. For more information, see OData geo-spatial functions in Azure Cognitive Search.
  - `search.in`, which compares a field or range variable with each value in a list of values. For more information, see OData search.in function in Azure Cognitive Search.
  - `search.ismatch` and `search.ismatchscoring`, which execute full-text search operations in a filter context. For more information, see OData full-text search functions in Azure Cognitive Search.
- Field paths or range variables of type `Edm.Boolean`. For example, if your index has a Boolean field called `IsEnabled` and you want to return all documents where this field is `true`, your filter expression can just be the name `IsEnabled`.
- Boolean expressions in parentheses. Using parentheses can help to explicitly determine the order of operations in a filter. For more information on the default precedence of the OData operators, see the next section.

Operator precedence in filters

If you write a filter expression with no parentheses around its sub-expressions, Azure Cognitive Search will evaluate it according to a set of operator precedence rules. These rules are based on which operators are used to combine sub-expressions. The following table lists groups of operators in order from highest to lowest precedence:

Group	Operator(s)

Group	Operator(s)
Logical operators	`not`
Comparison operators	`eq`, `ne`, `gt`, `lt`, `ge`, `le`
Logical operators	`and`
Logical operators	`or`

An operator that is higher in the above table will "bind more tightly" to its operands than other operators. For example, `and` is of higher precedence than `or`, and comparison operators are of higher precedence than either of them, so the following two expressions are equivalent:

odata-filter-expr

```
Rating gt 0 and Rating lt 3 or Rating gt 7 and Rating lt 10
((Rating gt 0) and (Rating lt 3)) or ((Rating gt 7) and (Rating lt 10))
```

The `not` operator has the highest precedence of all -- even higher than the comparison operators. That's why if you try to write a filter like this:

odata-filter-expr

```
not Rating gt 5
```

You'll get this error message:

text

```
Invalid expression: A unary operator with an incompatible type was
detected. Found operand type 'Edm.Int32' for operator kind 'Not'.
```

This error happens because the operator is associated with just the `Rating` field, which is of type `Edm.Int32`, and not with the entire comparison expression. The fix is to put the operand of `not` in parentheses:

odata-filter-expr

```
not (Rating gt 5)
```

Filter size limitations

There are limits to the size and complexity of filter expressions that you can send to Azure Cognitive Search. The limits are based roughly on the number of clauses in your filter expression. A good guideline is that if you have hundreds of clauses, you are at risk of exceeding the limit. We recommend designing your application in such a way that it doesn't generate filters of unbounded size.

> **Tip**
>
> Using **the search.in function** instead of long disjunctions of equality comparisons can help avoid the filter clause limit, since a function call counts as a single clause.

Examples

Find all hotels with at least one room with a base rate less than $200 that are rated at or above 4:

odata-filter-expr

```
    $filter=Rooms/any(room: room/BaseRate lt 200.0) and Rating ge 4
```

Find all hotels other than "Sea View Motel" that have been renovated since 2010:

odata-filter-expr

```
    $filter=HotelName ne 'Sea View Motel' and LastRenovationDate ge 2010-01-
01T00:00:00Z
```

Find all hotels that were renovated in 2010 or later. The datetime literal includes time zone information for Pacific Standard Time:

odata-filter-expr

```
    $filter=LastRenovationDate ge 2010-01-01T00:00:00-08:00
```

Find all hotels that have parking included and where all rooms are non-smoking:

odata-filter-expr

```
    $filter=ParkingIncluded and Rooms/all(room: not room/SmokingAllowed)
```

- OR -

odata-filter-expr

```
    $filter=ParkingIncluded eq true and Rooms/all(room: room/SmokingAllowed
eq false)
```

Find all hotels that are Luxury or include parking and have a rating of 5:

odata-filter-expr

```
    $filter=(Category eq 'Luxury' or ParkingIncluded eq true) and Rating eq
5
```

Find all hotels with the tag "wifi" in at least one room (where each room has tags stored in a `Collection(Edm.String)` field):

odata-filter-expr

```
    $filter=Rooms/any(room: room/Tags/any(tag: tag eq 'wifi'))
```

Find all hotels with any rooms:

odata-filter-expr

```
    $filter=Rooms/any()
```

Find all hotels that don't have rooms:

odata-filter-expr

```
    $filter=not Rooms/any()
```

Find all hotels within 10 kilometers of a given reference point (where `Location` is a field of type `Edm.GeographyPoint`):

odata-filter-expr

```
    $filter=geo.distance(Location, geography'POINT(-122.131577 47.678581)')
le 10
```

Find all hotels within a given viewport described as a polygon (where `Location` is a field of type Edm.GeographyPoint). The polygon must be closed, meaning the first and last point sets must be the same. Also, the points must be listed in counterclockwise order.

odata-filter-expr

```
    $filter=geo.intersects(Location, geography'POLYGON((-122.031577
47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577
47.578581))')
```

Find all hotels where the "Description" field is null. The field will be null if it was never set, or if it was explicitly set to null:

odata-filter-expr

```
    $filter=Description eq null
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel'). These phrases contain spaces, and space is a default delimiter. You can specify an alternative delimiter in single quotes as the third string parameter:

odata-filter-expr

```
    $filter=search.in(HotelName, 'Sea View motel,Budget hotel', ',')
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel' separated by '|'):

odata-filter-expr

```
    $filter=search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels where all rooms have the tag 'wifi' or 'tub':

odata-filter-expr

```
    $filter=Rooms/any(room: room/Tags/any(tag: search.in(tag, 'wifi, tub')))
```

Find a match on phrases within a collection, such as 'heated towel racks' or 'hairdryer included' in tags.

odata-filter-expr

```
    $filter=Rooms/any(room: room/Tags/any(tag: search.in(tag, 'heated towel
racks,hairdryer included', ',')))
```

Find documents with the word "waterfront". This filter query is identical to a search request with `search=waterfront`.

odata-filter-expr

```
$filter=search.ismatchscoring('waterfront')
```

Find documents with the word "hostel" and rating greater or equal to 4, or documents with the word "motel" and rating equal to 5. This request couldn't be expressed without the `search.ismatchscoring` function since it combines full-text search with filter operations using `or`.

odata-filter-expr

```
$filter=search.ismatchscoring('hostel') and rating ge 4 or
search.ismatchscoring('motel') and rating eq 5
```

Find documents without the word "luxury".

odata-filter-expr

```
$filter=not search.ismatch('luxury')
```

Find documents with the phrase "ocean view" or rating equal to 5. The `search.ismatchscoring` query will be executed only against fields `HotelName` and `Description`. Documents that matched only the second clause of the disjunction will be returned too -- hotels with `Rating` equal to 5. Those documents will be returned with score equal to zero to make it clear that they didn't match any of the scored parts of the expression.

odata-filter-expr

```
$filter=search.ismatchscoring('"ocean view"', 'Description,HotelName')
or Rating eq 5
```

Find hotels where the terms "hotel" and "airport" are no more than five words apart in the description, and where all rooms are non-smoking. This query uses the full Lucene query language.

odata-filter-expr

```
$filter=search.ismatch('"hotel airport"~5', 'Description', 'full',
'any') and not Rooms/any(room: room/SmokingAllowed)
```

Find documents that have a word that starts with the letters "lux" in the Description field. This query uses prefix search in combination with `search.ismatch`.

odata-filter-expr

```
$filter=search.ismatch('lux*', 'Description')
```

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

Documentation

OData expression syntax reference - Azure Cognitive Search

Formal grammar and syntax specification for OData expressions in Azure Cognitive Search queries.

OData select reference - Azure Cognitive Search

Syntax and language reference for explicit selection of fields to return in the search results of Azure Cognitive Search queries.

OData collection operator reference - Azure Cognitive Search

When creating filter expressions in Azure Cognitive Search queries, use "any" and "all" operators in lambda expressions when the filter is on a collection or complex collection field.

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search queries.

OData order-by reference - Azure Cognitive Search

Syntax and language reference documentation for using order-by in Azure Cognitive Search queries.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

OData full-text search function reference - Azure Cognitive Search

OData full-text search functions, search.ismatch and search.ismatchscoring, in Azure Cognitive Search queries.

Understanding OData collection filters - Azure Cognitive Search

Learn the mechanics of how OData collection filters work in Azure Cognitive Search queries, including limitations and behaviors unique to collections.

Show 5 more

Training

Learning path

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

OData $orderby syntax in Azure Cognitive Search

Article • 11/02/2022 • 2 minutes to read

In Azure Cognitive Search, the **$orderby** parameter specifies a custom sort order for search results. This article describes the OData syntax of **$orderby** and provides examples.

Field path construction and constants are described in the OData language overview in Azure Cognitive Search. For more information about sorting behaviors, see Ordering results.

Syntax

The **$orderby** parameter accepts a comma-separated list of up to 32 **order-by clauses**. The syntax of an order-by clause is described by the following EBNF (Extended Backus-Naur Form):

```
order_by_clause ::= (field_path | sortable_function) ('asc' | 'desc')?

sortable_function ::= geo_distance_call | 'search.score()'
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

Each clause has sort criteria, optionally followed by a sort direction (`asc` for ascending or `desc` for descending). If you don't specify a direction, the default is ascending. If there are null values in the field, null values appear first if the sort is `asc` and last if the sort is `desc`.

The sort criteria can either be the path of a `sortable` field or a call to either the geo.distance or the search.score functions.

For string fields, the default ASCII sort order and default Unicode sort order will be used. By default, sorting is case sensitive but you can use a normalizer to preprocess the text before sorting to change this behavior. You can also use an `asciifolding` normalizer to convert non-ASCII characters to their ASCII equivalent, if one exists.

If multiple documents have the same sort criteria and the `search.score` function isn't used (for example, if you sort by a numeric `Rating` field and three documents all have a rating of 4), ties will be broken by document score in descending order. When document scores are the same (for example, when there's no full-text search query specified in the request), then the relative ordering of the tied documents is indeterminate.

You can specify multiple sort criteria. The order of expressions determines the final sort order. For example, to sort descending by score, followed by Rating, the syntax would be `$orderby=search.score() desc,Rating desc`.

The syntax for `geo.distance` in **$orderby** is the same as it is in **$filter**. When using `geo.distance` in **$orderby**, the field to which it applies must be of type `Edm.GeographyPoint` and it must also be `sortable`.

The syntax for `search.score` in **$orderby** is `search.score()`. The function `search.score` doesn't take any parameters.

Examples

Sort hotels ascending by base rate:

odata-filter-expr

```
$orderby=BaseRate asc
```

Sort hotels descending by rating, then ascending by base rate (remember that ascending is the default):

odata-filter-expr

```
$orderby=Rating desc,BaseRate
```

Sort hotels descending by rating, then ascending by distance from the given coordinates:

odata-filter-expr

```
    $orderby=Rating desc,geo.distance(Location, geography'POINT(-122.131577
47.678581)') asc
```

Sort hotels in descending order by search.score and rating, and then in ascending order by distance from the given coordinates. Between two hotels with identical relevance scores and ratings, the closest one is listed first:

odata-filter-expr

```
    $orderby=search.score() desc,Rating desc,geo.distance(Location,
geography'POINT(-122.131577 47.678581)') asc
```

See also

- How to work with search results in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

🕮 Documentation

OData select reference - Azure Cognitive Search

Syntax and language reference for explicit selection of fields to return in the search results of Azure Cognitive Search queries.

OData expression syntax reference - Azure Cognitive Search

Formal grammar and syntax specification for OData expressions in Azure Cognitive Search queries.

OData collection operator reference - Azure Cognitive Search

When creating filter expressions in Azure Cognitive Search queries, use "any" and "all" operators in lambda expressions when the filter is on a collection or complex collection field.

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search queries.

OData full-text search function reference - Azure Cognitive Search

OData full-text search functions, search.ismatch and search.ismatchscoring, in Azure Cognitive Search queries.

OData filter reference - Azure Cognitive Search

OData language reference and full syntax used for creating filter expressions in Azure Cognitive Search queries.

OData logical operator reference - Azure Cognitive Search

Syntax and reference documentation for using OData logical operators, and, or, and not, in Azure Cognitive Search queries.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

Show 5 more

OData $select syntax in Azure Cognitive Search

Article • 07/19/2022 • 2 minutes to read

In Azure Cognitive Search, the **$select** parameter specifies which fields to include in search results. This article describes the OData syntax of **$select** and provides examples.

Field path construction and constants are described in the OData language overview in Azure Cognitive Search. For more information about search result composition, see How to work with search results in Azure Cognitive Search.

Syntax

The **$select** parameter determines which fields for each document are returned in the query result set. The following EBNF (Extended Backus-Naur Form) defines the grammar for the **$select** parameter:

```
select_expression ::= '*' | field_path(',' field_path)*

field_path ::= identifier('/'identifier)*
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

The **$select** parameter comes in two forms:

1. A single star (`*`), indicating that all retrievable fields should be returned, or
2. A comma-separated list of field paths, identifying which fields should be returned.

When using the second form, you may only specify retrievable fields in the list.

If you list a complex field without specifying its subfields explicitly, all retrievable subfields will be included in the query result set. For example, assume your index has an

`Address` field with `Street`, `City`, and `Country` subfields that are all retrievable. If you specify `Address` in **$select**, the query results will include all three subfields.

Examples

Include the `HotelId`, `HotelName`, and `Rating` top-level fields in the results, and include the `City` subfield of `Address`:

odata-filter-expr

```
$select=HotelId, HotelName, Rating, Address/City
```

An example result might look like this:

JSON

```
{
  "HotelId": "1",
  "HotelName": "Secret Point Motel",
  "Rating": 4,
  "Address": {
    "City": "New York"
  }
}
```

Include the `HotelName` top-level field in the results. Include all subfields of `Address`. Include the `Type` and `BaseRate` subfields of each object in the `Rooms` collection:

odata-filter-expr

```
$select=HotelName, Address, Rooms/Type, Rooms/BaseRate
```

An example result might look like this:

JSON

```
{
  "HotelName": "Secret Point Motel",
  "Rating": 4,
  "Address": {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY",
    "Country": "USA",
    "PostalCode": "10022"
  },
```

```
  "Rooms": [
    {
      "Type": "Budget Room",
      "BaseRate": 9.69
    },
    {
      "Type": "Budget Room",
      "BaseRate": 8.09
    }
  ]
}
```

Next steps

- How to work with search results in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

🕮 Documentation

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search queries.

OData collection operator reference - Azure Cognitive Search

When creating filter expressions in Azure Cognitive Search queries, use "any" and "all" operators in lambda expressions when the filter is on a collection or complex collection field.

Understanding OData collection filters - Azure Cognitive Search

Learn the mechanics of how OData collection filters work in Azure Cognitive Search queries, including limitations and behaviors unique to collections.

OData full-text search function reference - Azure Cognitive Search

OData full-text search functions, search.ismatch and search.ismatchscoring, in Azure Cognitive Search queries.

OData expression syntax reference - Azure Cognitive Search

Formal grammar and syntax specification for OData expressions in Azure Cognitive Search queries.

OData order-by reference - Azure Cognitive Search

Syntax and language reference documentation for using order-by in Azure Cognitive Search queries.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

OData geo-spatial function reference - Azure Cognitive Search

Syntax and reference documentation for using OData geo-spatial functions, geo.distance and geo.intersects, in Azure Cognitive Search queries.

Show 5 more

Training

Learning path

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

OData collection operators in Azure Cognitive Search - `any` and `all`

Article • 02/09/2023 • 3 minutes to read

When writing an OData filter expression to use with Azure Cognitive Search, it's often useful to filter on collection fields. You can achieve this using the `any` and `all` operators.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of an OData expression that uses `any` or `all`.

```
collection_filter_expression ::=
    field_path'/all(' lambda_expression ')'
    | field_path'/any(' lambda_expression ')'
    | field_path'/any()'

lambda_expression ::= identifier ':' boolean_expression
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

There are three forms of expression that filter collections.

- The first two iterate over a collection field, applying a predicate given in the form of a lambda expression to each element of the collection.
  - An expression using `all` returns `true` if the predicate is true for every element of the collection.
  - An expression using `any` returns `true` if the predicate is true for at least one element of the collection.

- The third form of collection filter uses `any` without a lambda expression to test whether a collection field is empty. If the collection has any elements, it returns `true`. If the collection is empty, it returns `false`.

A **lambda expression** in a collection filter is like the body of a loop in a programming language. It defines a variable, called the **range variable**, that holds the current element of the collection during iteration. It also defines another boolean expression that is the filter criteria to apply to the range variable for each element of the collection.

Examples

Match documents whose `tags` field contains exactly the string "wifi":

```text
tags/any(t: t eq 'wifi')
```

Match documents where every element of the `ratings` field falls between 3 and 5, inclusive:

```text
ratings/all(r: r ge 3 and r le 5)
```

Match documents where any of the geo coordinates in the `locations` field is within the given polygon:

```text
locations/any(loc: geo.intersects(loc, geography'POLYGON((-122.031577
47.578581, -122.031577 47.678581, -122.131577 47.678581, -122.031577
47.578581))'))
```

Match documents where the `rooms` field is empty:

```text
not rooms/any()
```

Match documents where (for all rooms) the `rooms/amenities` field contains "tv", and `rooms/baseRate` is less than 100:

```text
```

```
rooms/all(room: room/amenities/any(a: a eq 'tv') and room/baseRate lt 100.0)
```

Limitations

Not every feature of filter expressions is available inside the body of a lambda expression. The limitations differ depending on the data type of the collection field that you want to filter. The following table summarizes the limitations.

Data type	**Features allowed in lambda expressions with `any`**	**Features allowed in lambda expressions with `all`**
`Collection(Edm.ComplexType)`	Everything except `search.ismatch` and `search.ismatchscoring`	Same
`Collection(Edm.String)`	Comparisons with `eq` or `search.in` Combining sub-expressions with `or`	Comparisons with `ne` or `not search.in()` Combining sub-expressions with `and`
`Collection(Edm.Boolean)`	Comparisons with `eq` or `ne`	Same
`Collection(Edm.GeographyPoint)`	Using `geo.distance` with `lt` or `le` `geo.intersects` Combining sub-expressions with `or`	Using `geo.distance` with `gt` or `ge` `not geo.intersects(...)` Combining sub-expressions with `and`

Data type	Features allowed in lambda expressions with `any`	Features allowed in lambda expressions with `all`
`Collection(Edm.DateTimeOffset)`, `Collection(Edm.Double)`, `Collection(Edm.Int32)`, `Collection(Edm.Int64)`	Comparisons using `eq`, `ne`, `lt`, `gt`, `le`, or `ge` Combining comparisons with other sub-expressions using `or` Combining comparisons except `ne` with other sub-expressions using `and` Expressions using combinations of `and` and `or` in Disjunctive Normal Form (DNF)	Comparisons using `eq`, `ne`, `lt`, `gt`, `le`, or `ge` Combining comparisons with other sub-expressions using `and` Combining comparisons except `eq` with other sub-expressions using `or` Expressions using combinations of `and` and `or` in Conjunctive Normal Form (CNF)

For more details on these limitations as well as examples, see Troubleshooting collection filters in Azure Cognitive Search. For more in-depth information on why these limitations exist, see Understanding collection filters in Azure Cognitive Search.

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

Documentation

OData select reference - Azure Cognitive Search

Syntax and language reference for explicit selection of fields to return in the search results of Azure Cognitive Search queries.

OData expression syntax reference - Azure Cognitive Search

Formal grammar and syntax specification for OData expressions in Azure Cognitive Search queries.

OData full-text search function reference - Azure Cognitive Search

OData full-text search functions, search.ismatch and search.ismatchscoring, in Azure Cognitive Search

queries.

OData order-by reference - Azure Cognitive Search

Syntax and language reference documentation for using order-by in Azure Cognitive Search queries.

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search queries.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

Understanding OData collection filters - Azure Cognitive Search

Learn the mechanics of how OData collection filters work in Azure Cognitive Search queries, including limitations and behaviors unique to collections.

OData filter reference - Azure Cognitive Search

OData language reference and full syntax used for creating filter expressions in Azure Cognitive Search queries.

Show 5 more

OData comparison operators in Azure Cognitive Search - `eq`, `ne`, `gt`, `lt`, `ge`, and `le`

Article • 03/10/2022 • 5 minutes to read

The most basic operation in an OData filter expression in Azure Cognitive Search is to compare a field to a given value. Two types of comparison are possible -- equality comparison, and range comparison. You can use the following operators to compare a field to a constant value:

Equality operators:

- `eq`: Test whether a field is **equal to** a constant value
- `ne`: Test whether a field is **not equal to** a constant value

Range operators:

- `gt`: Test whether a field is **greater than** a constant value
- `lt`: Test whether a field is **less than** a constant value
- `ge`: Test whether a field is **greater than or equal to** a constant value
- `le`: Test whether a field is **less than or equal to** a constant value

You can use the range operators in combination with the logical operators to test whether a field is within a certain range of values. See the examples later in this article.

> **Note**
>
> If you prefer, you can put the constant value on the left side of the operator and the field name on the right side. For range operators, the meaning of the comparison is reversed. For example, if the constant value is on the left, `gt` would test whether the constant value is greater than the field. You can also use the comparison operators to compare the result of a function, such as `geo.distance`, with a value. For Boolean functions such as `search.ismatch`, comparing the result to `true` or `false` is optional.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of an OData expression that uses the comparison operators.

```
comparison_expression ::=
    variable_or_function comparison_operator constant |
    constant comparison_operator variable_or_function

variable_or_function ::= variable | function_call

comparison_operator ::= 'gt' | 'lt' | 'ge' | 'le' | 'eq' | 'ne'
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

There are two forms of comparison expressions. The only difference between them is whether the constant appears on the left- or right-hand-side of the operator. The expression on the other side of the operator must be a **variable** or a function call. A variable can be either a field name, or a range variable in the case of a lambda expression.

Data types for comparisons

The data types on both sides of a comparison operator must be compatible. For example, if the left side is a field of type `Edm.DateTimeOffset`, then the right side must be a date-time constant. Numeric data types are more flexible. You can compare variables and functions of any numeric type with constants of any other numeric type, with a few limitations, as described in the following table.

Variable or function type	Constant value type	Limitations
`Edm.Double`	`Edm.Double`	Comparison is subject to special rules for NaN
`Edm.Double`	`Edm.Int64`	Constant is converted to `Edm.Double`, resulting in a loss of precision for values of large magnitude

Variable or function type	Constant value type	Limitations
`Edm.Double`	`Edm.Int32`	n/a
`Edm.Int64`	`Edm.Double`	Comparisons to `NaN`, `-INF`, or `INF` are not allowed
`Edm.Int64`	`Edm.Int64`	n/a
`Edm.Int64`	`Edm.Int32`	Constant is converted to `Edm.Int64` before comparison
`Edm.Int32`	`Edm.Double`	Comparisons to `NaN`, `-INF`, or `INF` are not allowed
`Edm.Int32`	`Edm.Int64`	n/a
`Edm.Int32`	`Edm.Int32`	n/a

For comparisons that are not allowed, such as comparing a field of type `Edm.Int64` to `NaN`, the Azure Cognitive Search REST API will return an "HTTP 400: Bad Request" error.

> **Important**
>
> Even though numeric type comparisons are flexible, we highly recommend writing comparisons in filters so that the constant value is of the same data type as the variable or function to which it is being compared. This is especially important when mixing floating-point and integer values, where implicit conversions that lose precision are possible.

Special cases for `null` and `NaN`

When using comparison operators, it's important to remember that all non-collection fields in Azure Cognitive Search can potentially be `null`. The following table shows all the possible outcomes for a comparison expression where either side can be `null`:

Operator	Result when only the field or variable is `null`	Result when only the constant is `null`	Result when both the field or variable and the constant are `null`
`gt`	`false`	HTTP 400: Bad Request error	HTTP 400: Bad Request error
`lt`	`false`	HTTP 400: Bad Request error	HTTP 400: Bad Request error

Operator	Result when only the field or variable is `null`	Result when only the constant is `null`	Result when both the field or variable and the constant are `null`
`ge`	`false`	HTTP 400: Bad Request error	HTTP 400: Bad Request error
`le`	`false`	HTTP 400: Bad Request error	HTTP 400: Bad Request error
`eq`	`false`	`false`	`true`
`ne`	`true`	`true`	`false`

In summary, `null` is equal only to itself, and is not less or greater than any other value.

If your index has fields of type `Edm.Double` and you upload `NaN` values to those fields, you will need to account for that when writing filters. Azure Cognitive Search implements the IEEE 754 standard for handling `NaN` values, and comparisons with such values produce non-obvious results, as shown in the following table.

Operator	Result when at least one operand is `NaN`
`gt`	`false`
`lt`	`false`
`ge`	`false`
`le`	`false`
`eq`	`false`
`ne`	`true`

In summary, `NaN` is not equal to any value, including itself.

Comparing geo-spatial data

You can't directly compare a field of type `Edm.GeographyPoint` with a constant value, but you can use the `geo.distance` function. This function returns a value of type `Edm.Double`, so you can compare it with a numeric constant to filter based on the distance from constant geo-spatial coordinates. See the examples below.

Comparing string data

Strings can be compared in filters for exact matches using the `eq` and `ne` operators. These comparisons are case-sensitive.

Examples

Match documents where the `Rating` field is between 3 and 5, inclusive:

```text
Rating ge 3 and Rating le 5
```

Match documents where the `Location` field is less than 2 kilometers from the given latitude and longitude:

```text
geo.distance(Location, geography'POINT(-122.031577 47.578581)') lt 2.0
```

Match documents where the `LastRenovationDate` field is greater than or equal to January 1st, 2015, midnight UTC:

```text
LastRenovationDate ge 2015-01-01T00:00:00.000Z
```

Match documents where the `Details/Sku` field is not `null`:

```text
Details/Sku ne null
```

Match documents for hotels where at least one room has type "Deluxe Room", where the string of the `Rooms/Type` field matches the filter exactly:

```text
Rooms/any(room: room/Type eq 'Deluxe Room')
```

Next steps

- Filters in Azure Cognitive Search

- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

OData full-text search functions in Azure Cognitive Search - `search.ismatch` and `search.ismatchscoring`

Article • 03/10/2022 • 3 minutes to read

Azure Cognitive Search supports full-text search in the context of OData filter expressions via the `search.ismatch` and `search.ismatchscoring` functions. These functions allow you to combine full-text search with strict Boolean filtering in ways that are not possible just by using the top-level `search` parameter of the Search API.

> **Note**
>
> The `search.ismatch` and `search.ismatchscoring` functions are only supported in filters in the **Search API**. They are not supported in the **Suggest** or **Autocomplete** APIs.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of the `search.ismatch` and `search.ismatchscoring` functions:

```
search_is_match_call ::=
    'search.ismatch'('scoring')?'(' search_is_match_parameters ')'

search_is_match_parameters ::=
    string_literal(',' string_literal(',' query_type ',' search_mode)?)?

query_type ::= "'full'" | "'simple'"

search_mode ::= "'any'" | "'all'"
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**

See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

search.ismatch

The `search.ismatch` function evaluates a full-text search query as a part of a filter expression. The documents that match the search query will be returned in the result set. The following overloads of this function are available:

- `search.ismatch(search)`
- `search.ismatch(search, searchFields)`
- `search.ismatch(search, searchFields, queryType, searchMode)`

The parameters are defined in the following table:

Parameter name	Type	Description
`search`	`Edm.String`	The search query (in either simple or full Lucene query syntax).
`searchFields`	`Edm.String`	Comma-separated list of searchable fields to search in; defaults to all searchable fields in the index. When using fielded search in the `search` parameter, the field specifiers in the Lucene query override any fields specified in this parameter.
`queryType`	`Edm.String`	`'simple'` or `'full'`; defaults to `'simple'`. Specifies what query language was used in the `search` parameter.
`searchMode`	`Edm.String`	`'any'` or `'all'`, defaults to `'any'`. Indicates whether any or all of the search terms in the `search` parameter must be matched in order to count the document as a match. When using the Lucene Boolean operators in the `search` parameter, they will take precedence over this parameter.

All the above parameters are equivalent to the corresponding search request parameters in the Search API.

The `search.ismatch` function returns a value of type `Edm.Boolean`, which allows you to compose it with other filter sub-expressions using the Boolean logical operators.

> **Note**
>
> Azure Cognitive Search does not support using `search.ismatch` or `search.ismatchscoring` inside lambda expressions. This means it is not possible to

write filters over collections of objects that can correlate full-text search matches with strict filter matches on the same object. For more details on this limitation as well as examples, see **Troubleshooting collection filters in Azure Cognitive Search**. For more in-depth information on why this limitation exists, see **Understanding collection filters in Azure Cognitive Search**.

search.ismatchscoring

The `search.ismatchscoring` function, like the `search.ismatch` function, returns `true` for documents that match the full-text search query passed as a parameter. The difference between them is that the relevance score of documents matching the `search.ismatchscoring` query will contribute to the overall document score, while in the case of `search.ismatch`, the document score won't be changed. The following overloads of this function are available with parameters identical to those of `search.ismatch`:

- `search.ismatchscoring(search)`
- `search.ismatchscoring(search, searchFields)`
- `search.ismatchscoring(search, searchFields, queryType, searchMode)`

Both the `search.ismatch` and `search.ismatchscoring` functions can be used in the same filter expression.

Examples

Find documents with the word "waterfront". This filter query is identical to a search request with `search=waterfront`.

odata-filter-expr

```
    search.ismatchscoring('waterfront')
```

Find documents with the word "hostel" and rating greater or equal to 4, or documents with the word "motel" and rating equal to 5. Note, this request could not be expressed without the `search.ismatchscoring` function.

odata-filter-expr

```
    search.ismatchscoring('hostel') and Rating ge 4 or
search.ismatchscoring('motel') and Rating eq 5
```

Find documents without the word "luxury".

odata-filter-expr

```
    not search.ismatch('luxury')
```

Find documents with the phrase "ocean view" or rating equal to 5. The `search.ismatchscoring` query will be executed only against fields `HotelName` and `Rooms/Description`.

Documents that matched only the second clause of the disjunction will be returned too -- hotels with `Rating` equal to 5. To make it clear that those documents didn't match any of the scored parts of the expression, they will be returned with score equal to zero.

odata-filter-expr

```
    search.ismatchscoring('"ocean view"', 'Rooms/Description,HotelName') or
Rating eq 5
```

Find documents where the terms "hotel" and "airport" are within 5 words from each other in the description of the hotel, and where smoking is not allowed in at least some of the rooms. This query uses the full Lucene query language.

odata-filter-expr

```
    search.ismatch('"hotel airport"~5', 'Description', 'full', 'any') and
Rooms/any(room: not room/SmokingAllowed)
```

Find documents that have a word that starts with the letters "lux" in the Description field. This query uses prefix search in combination with `search.ismatch`.

odata-filter-expr

```
    search.ismatch('lux*', 'Description')
```

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

Additional resources

Documentation

OData search.in function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.in function in Azure Cognitive Search queries.

Understanding OData collection filters - Azure Cognitive Search

Learn the mechanics of how OData collection filters work in Azure Cognitive Search queries, including limitations and behaviors unique to collections.

OData search.score function reference - Azure Cognitive Search

Syntax and reference documentation for using the search.score function in Azure Cognitive Search queries.

Troubleshooting OData collection filters - Azure Cognitive Search

Learn approaches for resolving OData collection filter errors in Azure Cognitive Search queries.

Simple query syntax - Azure Cognitive Search

Reference for the simple query syntax used for full text search queries in Azure Cognitive Search.

OData language overview - Azure Cognitive Search

OData language overview for filters, select, and order-by for Azure Cognitive Search queries.

Filter on search results - Azure Cognitive Search

Apply filter criteria to include or exclude content before query execution in Azure Cognitive Search.

Lucene query syntax - Azure Cognitive Search

Reference for the full Lucene query syntax, as used in Azure Cognitive Search for wildcard, fuzzy search, RegEx, and other advanced query constructs.

Show 5 more

Training

Learning path

Microsoft Search fundamentals - Training

Learn about Microsoft Search including where users can search, the answers and results they'll see, and how you can tailor the search experience for your organization.

OData geo-spatial functions in Azure Cognitive Search - `geo.distance` and `geo.intersects`

Article • 03/10/2022 • 4 minutes to read

Azure Cognitive Search supports geo-spatial queries in OData filter expressions via the `geo.distance` and `geo.intersects` functions. The `geo.distance` function returns the distance in kilometers between two points, one being a field or range variable, and one being a constant passed as part of the filter. The `geo.intersects` function returns `true` if a given point is within a given polygon, where the point is a field or range variable and the polygon is specified as a constant passed as part of the filter.

The `geo.distance` function can also be used in the **$orderby** parameter to sort search results by distance from a given point. The syntax for `geo.distance` in **$orderby** is the same as it is in **$filter**. When using `geo.distance` in **$orderby**, the field to which it applies must be of type `Edm.GeographyPoint` and it must also be **sortable**.

> **Note**
>
> When using `geo.distance` in the **$orderby** parameter, the field you pass to the function must contain only a single geo-point. In other words, it must be of type `Edm.GeographyPoint` and not `Collection(Edm.GeographyPoint)`. It is not possible to sort on collection fields in Azure Cognitive Search.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of the `geo.distance` and `geo.intersects` functions, as well as the geo-spatial values on which they operate:

```
geo_distance_call ::=
    'geo.distance(' variable ',' geo_point ')'
    | 'geo.distance(' geo_point ',' variable ')'

geo_point ::= "geography'POINT(" lon_lat ")'"

lon_lat ::= float_literal ' ' float_literal
```

```
geo_intersects_call ::=
    'geo.intersects(' variable ',' geo_polygon ')'

/* You need at least four points to form a polygon, where the first and
last points are the same. */
geo_polygon ::=
    "geography'POLYGON((" lon_lat ',' lon_lat ',' lon_lat ',' lon_lat_list
"))'"

lon_lat_list ::= lon_lat(',' lon_lat)*
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

geo.distance

The `geo.distance` function takes two parameters of type `Edm.GeographyPoint` and returns an `Edm.Double` value that is the distance between them in kilometers. This differs from other services that support OData geo-spatial operations, which typically return distances in meters.

One of the parameters to `geo.distance` must be a geography point constant, and the other must be a field path (or a range variable in the case of a filter iterating over a field of type `Collection(Edm.GeographyPoint)`). The order of these parameters doesn't matter.

The geography point constant is of the form `geography'POINT(<longitude> <latitude>)'`, where the longitude and latitude are numeric constants.

> **Note**
>
> When using `geo.distance` in a filter, you must compare the distance returned by the function with a constant using `lt`, `le`, `gt`, or `ge`. The operators `eq` and `ne` are not supported when comparing distances. For example, this is a correct usage of `geo.distance`: `$filter=geo.distance(location, geography'POINT(-122.131577 47.678581)') le 5`.

geo.intersects

The `geo.intersects` function takes a variable of type `Edm.GeographyPoint` and a constant `Edm.GeographyPolygon` and returns an `Edm.Boolean` -- `true` if the point is within the bounds of the polygon, `false` otherwise.

The polygon is a two-dimensional surface stored as a sequence of points defining a bounding ring (see the examples below). The polygon needs to be closed, meaning the first and last point sets must be the same. Points in a polygon must be in counterclockwise order.

Geo-spatial queries and polygons spanning the 180th meridian

For many geo-spatial query libraries formulating a query that includes the 180th meridian (near the dateline) is either off-limits or requires a workaround, such as splitting the polygon into two, one on either side of the meridian.

In Azure Cognitive Search, geo-spatial queries that include 180-degree longitude will work as expected if the query shape is rectangular and your coordinates align to a grid layout along longitude and latitude (for example, `geo.intersects(location, geography'POLYGON((179 65, 179 66, -179 66, -179 65, 179 65))')`). Otherwise, for non-rectangular or unaligned shapes, consider the split polygon approach.

Geo-spatial functions and `null`

Like all other non-collection fields in Azure Cognitive Search, fields of type `Edm.GeographyPoint` can contain `null` values. When Azure Cognitive Search evaluates `geo.intersects` for a field that is `null`, the result will always be `false`. The behavior of `geo.distance` in this case depends on the context:

- In filters, `geo.distance` of a `null` field results in `null`. This means the document will not match because `null` compared to any non-null value evaluates to `false`.
- When sorting results using **$orderby**, `geo.distance` of a `null` field results in the maximum possible distance. Documents with such a field will sort lower than all others when the sort direction `asc` is used (the default), and higher than all others when the direction is `desc`.

Examples

Filter examples

Find all hotels within 10 kilometers of a given reference point (where location is a field of type `Edm.GeographyPoint`):

odata-filter-expr

```
    geo.distance(location, geography'POINT(-122.131577 47.678581)') le 10
```

Find all hotels within a given viewport described as a polygon (where location is a field of type `Edm.GeographyPoint`). Note that the polygon is closed (the first and last point sets must be the same) and the points must be listed in counterclockwise order.

odata-filter-expr

```
    geo.intersects(location, geography'POLYGON((-122.031577 47.578581,
-122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))')
```

Order-by examples

Sort hotels descending by `rating`, then ascending by distance from the given coordinates:

odata-filter-expr

```
    rating desc,geo.distance(location, geography'POINT(-122.131577
47.678581)') asc
```

Sort hotels in descending order by `search.score` and `rating`, and then in ascending order by distance from the given coordinates so that between two hotels with identical ratings, the closest one is listed first:

odata-filter-expr

```
    search.score() desc,rating desc,geo.distance(location,
geography'POINT(-122.131577 47.678581)') asc
```

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search

- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

OData logical operators in Azure Cognitive Search - `and`, `or`, `not`

Article • 03/10/2022 • 3 minutes to read

OData filter expressions in Azure Cognitive Search are Boolean expressions that evaluate to `true` or `false`. You can write a complex filter by writing a series of simpler filters and composing them using the logical operators from Boolean algebra:

- `and`: A binary operator that evaluates to `true` if both its left and right sub-expressions evaluate to `true`.
- `or`: A binary operator that evaluates to `true` if either one of its left or right sub-expressions evaluates to `true`.
- `not`: A unary operator that evaluates to `true` if its sub-expression evaluates to `false`, and vice-versa.

These, together with the collection operators any and all, allow you to construct filters that can express very complex search criteria.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of an OData expression that uses the logical operators.

```
logical_expression ::=
    boolean_expression ('and' | 'or') boolean_expression
    | 'not' boolean_expression
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

There are two forms of logical expressions: binary (`and`/`or`), where there are two sub-expressions, and unary (`not`), where there is only one. The sub-expressions can be Boolean expressions of any kind:

- Fields or range variables of type `Edm.Boolean`
- Functions that return values of type `Edm.Boolean`, such as `geo.intersects` or `search.ismatch`
- Comparison expressions, such as `rating gt 4`
- Collection expressions, such as `Rooms/any(room: room/Type eq 'Deluxe Room')`
- The Boolean literals `true` or `false`.
- Other logical expressions constructed using `and`, `or`, and `not`.

> ⓘ **Important**
>
> There are some situations where not all kinds of sub-expression can be used with `and`/`or`, particularly inside lambda expressions. See **OData collection operators in Azure Cognitive Search** for details.

Logical operators and `null`

Most Boolean expressions such as functions and comparisons cannot produce `null` values, and the logical operators cannot be applied to the `null` literal directly (for example, `x and null` is not allowed). However, Boolean fields can be `null`, so you need to be aware of how the `and`, `or`, and `not` operators behave in the presence of null. This is summarized in the following table, where `b` is a field of type `Edm.Boolean`:

Expression	Result when `b` is `null`
`b`	`false`
`not b`	`true`
`b eq true`	`false`
`b eq false`	`false`
`b eq null`	`true`
`b ne true`	`true`
`b ne false`	`true`
`b ne null`	`false`

Expression	Result when `b` is `null`
`b and true`	`false`
`b and false`	`false`
`b or true`	`true`
`b or false`	`false`

When a Boolean field `b` appears by itself in a filter expression, it behaves as if it had been written `b eq true`, so if `b` is `null`, the expression evaluates to `false`. Similarly, `not b` behaves like `not (b eq true)`, so it evaluates to `true`. In this way, `null` fields behave the same as `false`. This is consistent with how they behave when combined with other expressions using `and` and `or`, as shown in the table above. Despite this, a direct comparison to `false` (`b eq false`) will still evaluate to `false`. In other words, `null` is not equal to `false`, even though it behaves like it in Boolean expressions.

Examples

Match documents where the `rating` field is between 3 and 5, inclusive:

odata-filter-expr

```
    rating ge 3 and rating le 5
```

Match documents where all elements of the `ratings` field are less than 3 or greater than 5:

odata-filter-expr

```
    ratings/all(r: r lt 3 or r gt 5)
```

Match documents where the `location` field is within the given polygon, and the document does not contain the term "public".

odata-filter-expr

```
    geo.intersects(location, geography'POLYGON((-122.031577 47.578581,
-122.031577 47.678581, -122.131577 47.678581, -122.031577 47.578581))') and
not search.ismatch('public')
```

Match documents for hotels in Vancouver, Canada where there is a deluxe room with a base rate less than 160:

odata-filter-expr

```
    Address/City eq 'Vancouver' and Address/Country eq 'Canada' and
Rooms/any(room: room/Type eq 'Deluxe Room' and room/BaseRate lt 160)
```

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

OData `search.in` function in Azure Cognitive Search

Article • 03/10/2022 • 3 minutes to read

A common scenario in OData filter expressions is to check whether a single field in each document is equal to one of many possible values. For example, this is how some applications implement security trimming -- by checking a field containing one or more principal IDs against a list of principal IDs representing the user issuing the query. One way to write a query like this is to use the eq and or operators:

odata-filter-expr

```
group_ids/any(g: g eq '123' or g eq '456' or g eq '789')
```

However, there is a shorter way to write this, using the `search.in` function:

odata-filter-expr

```
group_ids/any(g: search.in(g, '123, 456, 789'))
```

> **Important**
>
> Besides being shorter and easier to read, using `search.in` also provides **performance benefits** and avoids certain **size limitations of filters** when there are hundreds or even thousands of values to include in the filter. For this reason, we strongly recommend using `search.in` instead of a more complex disjunction of equality expressions.

> **Note**
>
> Version 4.01 of the OData standard has recently introduced the **in operator**, which has similar behavior as the `search.in` function in Azure Cognitive Search. However, Azure Cognitive Search does not support this operator, so you must use the `search.in` function instead.

Syntax

The following EBNF (Extended Backus-Naur Form) defines the grammar of the `search.in` function:

```
search_in_call ::=
    'search.in(' variable ',' string_literal(',' string_literal)? ')'
```

An interactive syntax diagram is also available:

OData syntax diagram for Azure Cognitive Search

> **Note**
>
> See **OData expression syntax reference for Azure Cognitive Search** for the complete EBNF.

The `search.in` function tests whether a given string field or range variable is equal to one of a given list of values. Equality between the variable and each value in the list is determined in a case-sensitive fashion, the same way as for the `eq` operator. Therefore an expression like `search.in(myfield, 'a, b, c')` is equivalent to `myfield eq 'a' or myfield eq 'b' or myfield eq 'c'`, except that `search.in` will yield much better performance.

There are two overloads of the `search.in` function:

- `search.in(variable, valueList)`
- `search.in(variable, valueList, delimiters)`

The parameters are defined in the following table:

Parameter name	Type	Description
`variable`	`Edm.String`	A string field reference (or a range variable over a string collection field in the case where `search.in` is used inside an `any` or `all` expression).
`valueList`	`Edm.String`	A string containing a delimited list of values to match against the `variable` parameter. If the `delimiters` parameter is not specified, the default delimiters are space and comma.

Parameter name	Type	Description
`delimiters`	`Edm.String`	A string where each character is treated as a separator when parsing the `valueList` parameter. The default value of this parameter is `' ,'` which means that any values with spaces and/or commas between them will be separated. If you need to use separators other than spaces and commas because your values include those characters, you can specify alternate delimiters such as `'|'` in this parameter.

Performance of `search.in`

If you use `search.in`, you can expect sub-second response time when the second parameter contains a list of hundreds or thousands of values. There is no explicit limit on the number of items you can pass to `search.in`, although you are still limited by the maximum request size. However, the latency will grow as the number of values grows.

Examples

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel'. Phrases contain spaces, which is a default delimiter. You can specify an alternative delimiter in single quotes as the third string parameter:

odata-filter-expr

```
search.in(HotelName, 'Sea View motel,Budget hotel', ',')
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel' separated by '|'):

odata-filter-expr

```
search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels with rooms that have the tag 'wifi' or 'tub':

odata-filter-expr

```
Rooms/any(room: room/Tags/any(tag: search.in(tag, 'wifi, tub')))
```

Find a match on phrases within a collection, such as 'heated towel racks' or 'hairdryer included' in tags.

odata-filter-expr

```
    Rooms/any(room: room/Tags/any(tag: search.in(tag, 'heated towel
racks,hairdryer included', ',')))
```

Find all hotels without the tag 'motel' or 'cabin':

odata-filter-expr

```
    Tags/all(tag: not search.in(tag, 'motel, cabin'))
```

Next steps

- Filters in Azure Cognitive Search
- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

OData `search.score` function in Azure Cognitive Search

Article • 04/18/2023

When you send a query to Azure Cognitive Search without the **$orderby** parameter, the results that come back will be sorted in descending order by relevance score. Even when you do use **$orderby**, the relevance score is used to break ties by default. However, sometimes it's useful to use the relevance score as an initial sort criteria, and some other criteria as the tie-breaker. The example in this article demonstrates using the `search.score` function for sorting.

> **Note**
>
> The relevance score is computed by the similarity ranking algorithm, and the range varies depending on which algorithm you use. For more information, see **Relevance and scoring in Azure Cognitive Search**.

Syntax

The syntax for `search.score` in **$orderby** is `search.score()`. The function `search.score` doesn't take any parameters. It can be used with the `asc` or `desc` sort-order specifier, just like any other clause in the **$orderby** parameter. It can appear anywhere in the list of sort criteria.

Example

Sort hotels in descending order by `search.score` and `rating`, and then in ascending order by distance from the given coordinates so that between two hotels with identical ratings, the closest one is listed first:

odata-filter-expr

```
    search.score() desc,rating desc,geo.distance(location,
geography'POINT(-122.131577 47.678581)') asc
```

Next steps

- OData expression language overview for Azure Cognitive Search
- OData expression syntax reference for Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)

OData expression syntax reference for Azure Cognitive Search

Article • 10/23/2022 • 3 minutes to read

Azure Cognitive Search uses OData expressions as parameters throughout the API. Most commonly, OData expressions are used for the `$orderby` and `$filter` parameters. These expressions can be complex, containing multiple clauses, functions, and operators. However, even simple OData expressions like property paths are used in many parts of the Azure Cognitive Search REST API. For example, path expressions are used to refer to subfields of complex fields everywhere in the API, such as when listing subfields in a suggester, a scoring function, the `$select` parameter, or even fielded search in Lucene queries.

This article describes all these forms of OData expressions using a formal grammar. There is also an interactive diagram to help visually explore the grammar.

Formal grammar

We can describe the subset of the OData language supported by Azure Cognitive Search using an EBNF (Extended Backus-Naur Form) grammar. Rules are listed "top-down", starting with the most complex expressions, and breaking them down into more primitive expressions. At the top are the grammar rules that correspond to specific parameters of the Azure Cognitive Search REST API:

- $filter, defined by the `filter_expression` rule.
- $orderby, defined by the `order_by_expression` rule.
- $select, defined by the `select_expression` rule.
- Field paths, defined by the `field_path` rule. Field paths are used throughout the API. They can refer to either top-level fields of an index, or subfields with one or more complex field ancestors.

After the EBNF is a browsable syntax diagram that allows you to interactively explore the grammar and the relationships between its rules.

```
/* Top-level rules */

filter_expression ::= boolean_expression

order_by_expression ::= order_by_clause(',' order_by_clause)*
```

```
select_expression ::= '*' | field_path(',' field_path)*

field_path ::= identifier('/'identifier)*


/* Shared base rules */

identifier ::= [a-zA-Z_][a-zA-Z_0-9]*


/* Rules for $orderby */

order_by_clause ::= (field_path | sortable_function) ('asc' | 'desc')?

sortable_function ::= geo_distance_call | 'search.score()'


/* Rules for $filter */

boolean_expression ::=
    collection_filter_expression
    | logical_expression
    | comparison_expression
    | boolean_literal
    | boolean_function_call
    | '(' boolean_expression ')'
    | variable

/* This can be a range variable in the case of a lambda, or a field path. */
variable ::= identifier | field_path

collection_filter_expression ::=
    field_path'/all(' lambda_expression ')'
    | field_path'/any(' lambda_expression ')'
    | field_path'/any()'

lambda_expression ::= identifier ':' boolean_expression

logical_expression ::=
    boolean_expression ('and' | 'or') boolean_expression
    | 'not' boolean_expression

comparison_expression ::=
    variable_or_function comparison_operator constant |
    constant comparison_operator variable_or_function

variable_or_function ::= variable | function_call

comparison_operator ::= 'gt' | 'lt' | 'ge' | 'le' | 'eq' | 'ne'


/* Rules for constants and literals */

constant ::=
```

```
    string_literal
    | date_time_offset_literal
    | integer_literal
    | float_literal
    | boolean_literal
    | 'null'

string_literal ::= "'"([^'] | "''")*"'"

date_time_offset_literal ::= date_part'T'time_part time_zone

date_part ::= year'-'month'-'day

time_part ::= hour':'minute(':'second('.'fractional_seconds)?)?

zero_to_fifty_nine ::= [0-5]digit

digit ::= [0-9]

year ::= digit digit digit digit

month ::= '0'[1-9] | '1'[0-2]

day ::= '0'[1-9] | [1-2]digit | '3'[0-1]

hour ::= [0-1]digit | '2'[0-3]

minute ::= zero_to_fifty_nine

second ::= zero_to_fifty_nine

fractional_seconds ::= integer_literal

time_zone ::= 'Z' | sign hour':'minute

sign ::= '+' | '-'

/* In practice integer literals are limited in length to the precision of
the corresponding EDM data type. */
integer_literal ::= sign? digit+

float_literal ::=
    sign? whole_part fractional_part? exponent?
    | 'NaN'
    | '-INF'
    | 'INF'

whole_part ::= integer_literal

fractional_part ::= '.'integer_literal

exponent ::= 'e' sign? integer_literal

boolean_literal ::= 'true' | 'false'
```

```
/* Rules for functions */

function_call ::=
    geo_distance_call |
    boolean_function_call

geo_distance_call ::=
    'geo.distance(' variable ',' geo_point ')'
    | 'geo.distance(' geo_point ',' variable ')'

geo_point ::= "geography'POINT(" lon_lat ")'"

lon_lat ::= float_literal ' ' float_literal

boolean_function_call ::=
    geo_intersects_call |
    search_in_call |
    search_is_match_call

geo_intersects_call ::=
    'geo.intersects(' variable ',' geo_polygon ')'

/* You need at least four points to form a polygon, where the first and
last points are the same. */
geo_polygon ::=
    "geography'POLYGON((" lon_lat ',' lon_lat ',' lon_lat ',' lon_lat_list
"))'"

lon_lat_list ::= lon_lat(',' lon_lat)*

search_in_call ::=
    'search.in(' variable ',' string_literal(',' string_literal)? ')'

/* Note that it is illegal to call search.ismatch or search.ismatchscoring
from inside a lambda expression. */
search_is_match_call ::=
    'search.ismatch'('scoring')?'(' search_is_match_parameters ')'

search_is_match_parameters ::=
    string_literal(',' string_literal(',' query_type ',' search_mode)?)?

query_type ::= "'full'" | "'simple'"

search_mode ::= "'any'" | "'all'"
```

Syntax diagram

To visually explore the OData language grammar supported by Azure Cognitive Search, try the interactive syntax diagram:

OData syntax diagram for Azure Cognitive Search

See also

- Filters in Azure Cognitive Search
- Search Documents (Azure Cognitive Search REST API)
- Lucene query syntax
- Simple query syntax in Azure Cognitive Search

Simple query syntax in Azure Cognitive Search

Article • 12/19/2022 • 9 minutes to read

Azure Cognitive Search implements two Lucene-based query languages: Simple Query Parser and the Lucene Query Parser. The simple parser is more flexible and will attempt to interpret a request even if it's not perfectly composed. Because it's flexible, it's the default for queries in Azure Cognitive Search.

Query syntax for either parser applies to query expressions passed in the "search" parameter of a Search Documents (REST API) request, not to be confused with the OData syntax used for the "$filter" and "$orderby" expressions in the same request. OData parameters have different syntax and rules for constructing queries, escaping strings, and so on.

Although the simple parser is based on the Apache Lucene Simple Query Parser class, its implementation in Cognitive Search excludes fuzzy search. If you need fuzzy search, consider the alternative full Lucene query syntax instead.

Example (simple syntax)

This example shows a simple query, distinguished by `"queryType": "simple"` and valid syntax. Although query type is set below, it's the default and can be omitted unless you're reverting from an alternative type. The following example is a search over independent terms, with a requirement that all matching documents include "pool".

HTTP

```
POST https://{{service-name}}.search.windows.net/indexes/hotel-rooms-sample/docs/search?api-version=2020-06-30
{
  "queryType": "simple",
  "search": "budget hotel +pool",
  "searchMode": "all"
}
```

The "searchMode" parameter is relevant in this example. Whenever boolean operators are on the query, you should generally set `"searchMode=all"` to ensure that *all* of the criteria is matched. Otherwise, you can use the default `"searchMode=any"` that favors recall over precision.

For more examples, see Simple query syntax examples. For details about the query request and parameters, see Search Documents (REST API).

Keyword search on terms and phrases

Strings passed to the "search" parameter can include terms or phrases in any supported language, boolean operators, precedence operators, wildcard or prefix characters for "starts with" queries, escape characters, and URL encoding characters. The "search" parameter is optional. Unspecified, search (`search=*` or `search=" "`) returns the top 50 documents in arbitrary (unranked) order.

- A *term search* is a query of one or more terms, where any of the terms are considered a match.
- A *phrase search* is an exact phrase enclosed in quotation marks `" "`. For example, while `Roach Motel` (without quotes) would search for documents containing `Roach` and/or `Motel` anywhere in any order, `"Roach Motel"` (with quotes) will only match documents that contain that whole phrase together and in that order (lexical analysis still applies).

  Depending on your search client, you might need to escape the quotation marks in a phrase search. For example, in Postman in a POST request, a phrase search on `"Roach Motel"` in the request body would be specified as `"\"Roach Motel\""`.

By default, all strings passed in the "search" parameter undergo lexical analysis. Make sure you understand the tokenization behavior of the analyzer you're using. Often, when query results are unexpected, the reason can be traced to how terms are tokenized at query time. You can test tokenization on specific strings to confirm the output.

Any text input with one or more terms is considered a valid starting point for query execution. Azure Cognitive Search will match documents containing any or all of the terms, including any variations found during analysis of the text.

As straightforward as this sounds, there's one aspect of query execution in Azure Cognitive Search that *might* produce unexpected results, increasing rather than decreasing search results as more terms and operators are added to the input string. Whether this expansion actually occurs depends on the inclusion of a NOT operator, combined with a "searchMode" parameter setting that determines how NOT is interpreted in terms of AND or OR behaviors. For more information, see the NOT operator under Boolean operators.

Boolean operators

You can embed Boolean operators in a query string to improve the precision of a match. In the simple syntax, boolean operators are character-based. Text operators, such as the word AND, aren't supported.

Character	Example	Usage
`+`	`pool + ocean`	An AND operation. For example, `pool + ocean` stipulates that a document must contain both terms.
`|`	`pool | ocean`	An OR operation finds a match when either term is found. In the example, the query engine will return match on documents containing either `pool` or `ocean` or both. Because OR is the default conjunction operator, you could also leave it out, such that `pool ocean` is the equivalent of `pool | ocean`.
`-`	`pool - ocean`	A NOT operation returns matches on documents that exclude the term. The `searchMode` parameter on a query request controls whether a term with the NOT operator is ANDed or ORed with other terms in the query (assuming there's no boolean operators on the other terms). Valid values include `any` or `all`. `searchMode=any` increases the recall of queries by including more results, and by default `-` will be interpreted as "OR NOT". For example, `pool - ocean` will match documents that either contain the term `pool` or those that don't contain the term `ocean`. `searchMode=all` increases the precision of queries by including fewer results, and by default `-` will be interpreted as "AND NOT". For example, with `searchMode=any`, the query `pool - ocean` will match documents that contain the term "pool" and all documents that don't contain the term "ocean". This is arguably a more intuitive behavior for the `-` operator. Therefore, you should consider using `searchMode=all` instead of `searchMode=any` if you want to optimize searches for precision instead of recall, *and* Your users frequently use the `-` operator in searches. When deciding on a `searchMode` setting, consider the user interaction patterns for queries in various applications. Users who are searching for information are more likely to include an operator in a query, as opposed to e-commerce sites that have more built-in navigation structures.

Prefix queries

For "starts with" queries, add a suffix operator (`*`) as the placeholder for the remainder of a term. A prefix query must begin with at least one alphanumeric character before

you can add the suffix operator.

Character	Example	Usage
`*`	`lingui*` will match on "linguistic" or "linguini"	The asterisk (`*`) represents one or more characters of arbitrary length, ignoring case.

Similar to filters, a prefix query looks for an exact match. As such, there's no relevance scoring (all results receive a search score of 1.0). Be aware that prefix queries can be slow, especially if the index is large and the prefix consists of a small number of characters. An alternative methodology, such as edge n-gram tokenization, might perform faster. Terms using prefix search can't be longer than 1000 characters.

Simple syntax supports prefix matching only. For suffix or infix matching against the end or middle of a term, use the full Lucene syntax for wildcard search.

Escaping search operators

In the simple syntax, search operators include these characters: `+ | " ( ) ' \`

If any of these characters are part of a token in the index, escape it by prefixing it with a single backslash (`\`) in the query. For example, suppose you used a custom analyzer for whole term tokenization, and your index contains the string "Luxury+Hotel". To get an exact match on this token, insert an escape character: `search=luxury\+hotel`.

To make things simple for the more typical cases, there are two exceptions to this rule where escaping isn't needed:

- The NOT operator `-` only needs to be escaped if it's the first character after a whitespace. If the `-` appears in the middle (for example, in `3352CDD0-EF30-4A2E-A512-3B30AF40F3FD`), you can skip escaping.
- The suffix operator `*` only needs to be escaped if it's the last character before a whitespace. If the `*` appears in the middle (for example, in `4*4=16`), no escaping is needed.

> ⓘ **Note**
>
> By default, the standard analyzer will delete and break words on hyphens, whitespace, ampersands, and other characters during **lexical analysis**. If you require special characters to remain in the query string, you might need an analyzer that preserves them in the index. Some choices include Microsoft natural **language**

analyzers, which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see **Partial terms, patterns, and special characters**.

Encoding unsafe and reserved characters in URLs

Ensure all unsafe and reserved characters are encoded in a URL. For example, '#' is an unsafe character because it's a fragment/anchor identifier in a URL. The character must be encoded to `%23` if used in a URL. '&' and '=' are examples of reserved characters as they delimit parameters and specify values in Azure Cognitive Search. For more information, see RFC1738: Uniform Resource Locators (URL).

Unsafe characters are `" ` < > # % { } | \ ^ ~ [ ]`. Reserved characters are `; / ? : @ = + &`.

Special characters

Special characters can range from currency symbols like '$' or '€', to emojis. Many analyzers, including the default standard analyzer, will exclude special characters during indexing, which means they won't be represented in your index.

If you need special character representation, you can assign an analyzer that preserves them:

- The "whitespace" analyzer considers any character sequence separated by white spaces as tokens (so the '❤' emoji would be considered a token).
- A language analyzer, such as the Microsoft English analyzer ("en.microsoft"), would take the '$' or '€' string as a token.

For confirmation, you can test an analyzer to see what tokens are generated for a given string. As you might expect, you might not get full tokenization from a single analyzer. A workaround is to create multiple fields that contain the same content, but with different analyzer assignments (for example,"description_en", "description_fr", and so forth for language analyzers).

When using Unicode characters, make sure symbols are properly escaped in the query url (for instance for '❤' would use the escape sequence `%E2%9D%A4+`). Postman does this translation automatically.

Precedence (grouping)

You can use parentheses to create subqueries, including operators within the parenthetical statement. For example, `motel+(wifi|luxury)` will search for documents containing the "motel" term and either "wifi" or "luxury" (or both).

Query size limits

If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

- For GET, the length of the URL can't exceed 8 KB.
- For POST (and any other request), where the body of the request includes `search` and other parameters such as `filter` and `orderby`, the maximum size is 16 MB. Additional limits include:
  - The maximum length of the search clause is 100,000 characters.
  - The maximum number of clauses in `search` (expressions separated by AND or OR) is 1024.
  - The maximum search term size is 1000 characters for prefix search.
  - There's also a limit of approximately 32 KB on the size of any individual term in a query.

For more information on query limits, see API request limits.

Next steps

If you'll be constructing queries programmatically, review Full text search in Azure Cognitive Search to understand the stages of query processing and the implications of text analysis.

You can also review the following articles to learn more about query construction:

- Query examples for simple search
- Query examples for full Lucene search
- Search Documents REST API
- Lucene query syntax
- Filter and Select (OData) expression syntax

Lucene query syntax in Azure Cognitive Search

Article • 10/28/2022 • 12 minutes to read

When creating queries in Azure Cognitive Search, you can opt for the full Lucene Query Parser syntax for specialized query forms: wildcard, fuzzy search, proximity search, regular expressions. Much of the Lucene Query Parser syntax is implemented intact in Azure Cognitive Search, except for *range searches, which are constructed through `$filter` expressions.

To use full Lucene syntax, you'll set the queryType to "full" and pass in a query expression patterned for wildcard, fuzzy search, or one of the other query forms supported by the full syntax. In REST, query expressions are provided in the `search` parameter of a Search Documents (REST API) request.

Example (full syntax)

The following example is a search request constructed using the full syntax. This particular example shows in-field search and term boosting. It looks for hotels where the category field contains the term "budget". Any documents containing the phrase "recently renovated" are ranked higher as a result of the term boost value (3).

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
  "queryType": "full",
  "search": "category:budget AND \"recently renovated\"^3",
  "searchMode": "all"
}
```

While not specific to any query type, the `searchMode` parameter is relevant in this example. Whenever operators are on the query, you should generally set `searchMode=all` to ensure that *all* of the criteria is matched.

For more examples, see Lucene query syntax examples. For details about the query request and parameters, including searchMode, see Search Documents (REST API).

Syntax fundamentals

The following syntax fundamentals apply to all queries that use the Lucene syntax.

Operator evaluation in context

Placement determines whether a symbol is interpreted as an operator or just another character in a string.

For example, in Lucene full syntax, the tilde (~) is used for both fuzzy search and proximity search. When placed after a quoted phrase, ~ invokes proximity search. When placed at the end of a term, ~ invokes fuzzy search.

Within a term, such as "business~analyst", the character isn't evaluated as an operator. In this case, assuming the query is a term or phrase query, full text search with lexical analysis strips out the ~ and breaks the term "business~analyst" in two: business OR analyst.

The example above is the tilde (~), but the same principle applies to every operator.

Escaping special characters

In order to use any of the search operators as part of the search text, escape the character by prefixing it with a single backslash (`\`). For example, for a wildcard search on `https://`, where `://` is part of the query string, you would specify `search=https\:\/\/*`. Similarly, an escaped phone number pattern might look like this `\+1 \(800\) 642\-7676`.

Special characters that require escaping include the following:
`+ - & | ! ( ) { } [ ] ^ " ~ * ? : \ /`

> **Note**
>
> Although escaping keeps tokens together, **lexical analysis** during indexing may strip them out. For example, the standard Lucene analyzer will break words on hyphens, whitespace, and other characters. If you require special characters in the query string, you might need an analyzer that preserves them in the index. Some choices include Microsoft natural **language analyzers**, which preserves hyphenated words, or a custom analyzer for more complex patterns. For more information, see **Partial terms, patterns, and special characters**.

Encoding unsafe and reserved characters in URLs

Ensure all unsafe and reserved characters are encoded in a URL. For example, '#' is an unsafe character because it's a fragment/anchor identifier in a URL. The character must be encoded to `%23` if used in a URL. '&' and '=' are examples of reserved characters as they delimit parameters and specify values in Azure Cognitive Search. See RFC1738: Uniform Resource Locators (URL) for more details.

Unsafe characters are `" ` < > # % { } | \ ^ ~ [ ]`. Reserved characters are `; / ? : @ = + &`.

Boolean operators

You can embed Boolean operators in a query string to improve the precision of a match. The full syntax supports text operators in addition to character operators. Always specify text boolean operators (AND, OR, NOT) in all caps.

Text operator	Character	Example	Usage
AND	`+`	`wifi AND luxury`	Specifies terms that a match must contain. In the example, the query engine will look for documents containing both `wifi` and `luxury`. The plus character (`+`) can also be used directly in front of a term to make it required. For example, `+wifi +luxury` stipulates that both terms must appear somewhere in the field of a single document.
OR	(none) [1]	`wifi OR luxury`	Finds a match when either term is found. In the example, the query engine will return match on documents containing either `wifi` or `luxury` or both. Because OR is the default conjunction operator, you could also leave it out, such that `wifi luxury` is the equivalent of `wifi OR luxury`.
NOT	`!`, `-`	`wifi -luxury`	Returns matches on documents that exclude the term. For example, `wifi -luxury` will search for documents that have the `wifi` term but not `luxury`. It's important to note that the NOT operator (`NOT`, `!`, or `-`) behaves differently in full syntax than it does in simple syntax. In full syntax, negations will always be ANDed onto the query such that `wifi -luxury` is interpreted as "wifi AND NOT luxury" regardless of if the `searchMode` parameter is set to `any` or `all`. This gives you a more intuitive behavior for negations by default. A single negation such as the query `-luxury` isn't allowed in full search syntax and will always return an empty result set.

[1] The `|` character isn't supported for OR operations.

Fielded search

You can define a fielded search operation with the `fieldName:searchExpression` syntax, where the search expression can be a single word or a phrase, or a more complex expression in parentheses, optionally with Boolean operators. Some examples include the following:

- genre:jazz NOT history
- artists:("Miles Davis" "John Coltrane")

Be sure to put multiple strings within quotation marks if you want both strings to be evaluated as a single entity, in this case searching for two distinct artists in the `artists` field.

The field specified in `fieldName:searchExpression` must be a `searchable` field. See Create Index for details on how index attributes are used in field definitions.

> **Note**
>
> When using fielded search expressions, you do not need to use the `searchFields` parameter because each fielded search expression has a field name explicitly specified. However, you can still use the `searchFields` parameter if you want to run a query where some parts are scoped to a specific field, and the rest could apply to several fields. For example, the query `search=genre:jazz NOT history&searchFields=description` would match `jazz` only to the `genre` field, while it would match `NOT history` with the `description` field. The field name provided in `fieldName:searchExpression` always takes precedence over the `searchFields` parameter, which is why in this example, we do not need to include `genre` in the `searchFields` parameter.

Fuzzy search

A fuzzy search finds matches in terms that have a similar construction, expanding a term up to the maximum of 50 terms that meet the distance criteria of two or less. For more information, see Fuzzy search.

To do a fuzzy search, use the tilde "~" symbol at the end of a single word with an optional parameter, a number between 0 and 2 (default), that specifies the edit distance. For example, "blue~" or "blue~1" would return "blue", "blues", and "glue".

Fuzzy search can only be applied to terms, not phrases, but you can append the tilde to each term individually in a multi-part name or phrase. For example, "Unviersty~ of~ "Wshington~" would match on "University of Washington".

Proximity search

Proximity searches are used to find terms that are near each other in a document. Insert a tilde "~" symbol at the end of a phrase followed by the number of words that create the proximity boundary. For example, `"hotel airport"~5` will find the terms "hotel" and "airport" within five words of each other in a document.

Term boosting

Term boosting refers to ranking a document higher if it contains the boosted term, relative to documents that don't contain the term. This differs from scoring profiles in that scoring profiles boost certain fields, rather than specific terms.

The following example helps illustrate the differences. Suppose that there's a scoring profile that boosts matches in a certain field, say *genre* in the musicstoreindex example. Term boosting could be used to further boost certain search terms higher than others. For example, `rock^2 electronic` will boost documents that contain the search terms in the genre field higher than other searchable fields in the index. Further, documents that contain the search term *rock* will be ranked higher than the other search term *electronic* as a result of the term boost value (2).

To boost a term, use the caret, "^", symbol with a boost factor (a number) at the end of the term you're searching. You can also boost phrases. The higher the boost factor, the more relevant the term will be relative to other search terms. By default, the boost factor is 1. Although the boost factor must be positive, it can be less than 1 (for example, 0.20).

Regular expression search

A regular expression search finds a match based on patterns that are valid under Apache Lucene, as documented in the RegExp class. In Azure Cognitive Search, a regular expression is enclosed between forward slashes `/`.

For example, to find documents containing "motel" or "hotel", specify `/[mh]otel/`. Regular expression searches are matched against single words.

Some tools and languages impose other escape character requirements. For JSON, strings that include a forward slash are escaped with a backward slash: "microsoft.com/azure/" becomes `search=/.*microsoft.com\/azure\/.*/` where `search=/.* <string-placeholder>.*/` sets up the regular expression, and `microsoft.com\/azure\/` is the string with an escaped forward slash.

Two common symbols in regex queries are `.` and `*`. A `.` matches any one character and a `*` matches the previous character zero or more times. For example, `/be./` will match the terms "bee" and "bet" while `/be*/` would match "be", "bee", and "beee" but not "bet". Together, `.*` allow you to match any series of characters so `/be.*/` would match any term that starts with "be" such as "better".

Wildcard search

You can use generally recognized syntax for multiple (`*`) or single (`?`) character wildcard searches. Full Lucene syntax supports prefix, infix, and suffix matching.

Note the Lucene query parser supports the use of these symbols with a single term, and not a phrase.

Affix type	Description and examples
prefix	Term fragment comes before `*` or `?`. For example, a query expression of `search=alpha*` returns "alphanumeric" or "alphabetical". Prefix matching is supported in both simple and full syntax.
suffix	Term fragment comes after `*` or `?`, with a forward slash to delimit the construct. For example, `search=/.*numeric/` returns "alphanumeric".
infix	Term fragments enclose `*` or `?`. For example, `search=non*al` returns "nonnumerical" and "nonsensical".

You can combine operators in one expression. For example, `980?2*` matches on "98072-1222" and "98052-1234", where `?` matches on a single (required) character, and `*` matches on characters of an arbitrary length that follow.

Suffix matching requires the regular expression forward slash `/` delimiters. Generally, you can't use a `*` or `?` symbol as the first character of a term, without the `/`. It's also important to note that the `*` will behave differently when used outside of regex queries.

Outside or the regex forward slash `/` delimiters, the `*` is a wildcard character and will match any series of characters much like `.*` in regex. As an example, `search=/non.*al/` will produce the same result set as `search=non*al`.

> **Note**
>
> As a rule, pattern matching is slow so you might want to explore alternative methods, such as edge n-gram tokenization that creates tokens for sequences of characters in a term. With n-gram tokenization, the index will be larger, but queries might execute faster, depending on the pattern construction and the length of strings you are indexing. For more information, see **Partial term search and patterns with special characters**.

Impact of an analyzer on wildcard queries

During query parsing, queries that are formulated as prefix, suffix, wildcard, or regular expressions are passed as-is to the query tree, bypassing lexical analysis. Matches will only be found if the index contains the strings in the format your query specifies. In most cases, you'll need an analyzer during indexing that preserves string integrity so that partial term and pattern matching succeeds. For more information, see Partial term search in Azure Cognitive Search queries.

Consider a situation where you may want the search query 'terminat*' to return results that contain terms such as 'terminate', 'termination' and 'terminates'.

If you were to use the en.lucene (English Lucene) analyzer, it would apply aggressive stemming of each term. For example, 'terminate', 'termination', 'terminates' will all be tokenized down to the token 'termi' in your index. On the other side, terms in queries using wildcards or fuzzy search aren't analyzed at all., so there would be no results that would match the 'terminat*' query.

On the other side, the Microsoft analyzers (in this case, the en.microsoft analyzer) are a bit more advanced and use lemmatization instead of stemming. This means that all generated tokens should be valid English words. For example, 'terminate', 'terminates' and 'termination' will mostly stay whole in the index, and would be a preferable choice for scenarios that depend a lot on wildcards and fuzzy search.

Scoring wildcard and regex queries

Azure Cognitive Search uses frequency-based scoring (BM25) for text queries. However, for wildcard and regex queries where scope of terms can potentially be broad, the frequency factor is ignored to prevent the ranking from biasing towards matches from rarer terms. All matches are treated equally for wildcard and regex searches.

Special characters

In some circumstances, you may want to search for a special character, like an '❤' emoji or the '€' sign. In such cases, make sure that the analyzer you use doesn't filter those characters out. The standard analyzer bypasses many special characters, excluding them from your index.

Analyzers that will tokenize special characters include the "whitespace" analyzer, which takes into consideration any character sequences separated by whitespaces as tokens (so the "❤" string would be considered a token). Also, a language analyzer like the Microsoft English analyzer ("en.microsoft"), would take the "€" string as a token. You can test an analyzer to see what tokens it generates for a given query.

When using Unicode characters, make sure symbols are properly escaped in the query url (for instance for "❤" would use the escape sequence `%E2%9D%A4+`). Postman does this translation automatically.

Precedence (grouping)

You can use parentheses to create subqueries, including operators within the parenthetical statement. For example, `motel+(wifi|luxury)` will search for documents containing the "motel" term and either "wifi" or "luxury" (or both).

Field grouping is similar but scopes the grouping to a single field. For example, `hotelAmenities:(gym+(wifi|pool))` searches the field "hotelAmenities" for "gym" and "wifi", or "gym" and "pool".

Query size limits

Azure Cognitive Search imposes limits on query size and composition because unbounded queries can destabilize your search service. There are limits on query size and composition (the number of clauses). Limits also exist for the length of prefix search and for the complexity of regex search and wildcard search. If your application generates search queries programmatically, we recommend designing it in such a way that it doesn't generate queries of unbounded size.

For more information on query limits, see API request limits.

See also

- Query examples for simple search
- Query examples for full Lucene search
- Search Documents
- OData expression syntax for filters and sorting
- Simple query syntax in Azure Cognitive Search

moreLikeThis (preview) in Azure Cognitive Search

Article • 09/16/2022 • 2 minutes to read

> **Important**
>
> This feature is in public preview under **Supplemental Terms of Use**. The **preview REST API** supports this feature.

`moreLikeThis=[key]` is a query parameter in the Search Documents API that finds documents similar to the document specified by the document key. When a search request is made with `moreLikeThis`, a query is generated with search terms extracted from the given document that describe that document best. The generated query is then used to make the search request. The `moreLikeThis` parameter can't be used with the search parameter, `search=[string]`.

By default, the contents of all top-level searchable fields are considered. If you want to specify particular fields instead, you can use the `searchFields` parameter.

The `moreLikeThis` parameter isn't supported for complex types and the presence of complex types will impact your query logic. If your index is a complex type, you must set `searchFields` to the top-level searchable fields over which `moreLikeThis` iterates. For example, if the index has a searchable `field1` of type `Edm.String`, and `field2` that's a complex type with searchable subfields, the value of `searchFields` must be set to `field1` to exclude `field2`.

Examples

All following examples use the hotels sample from Quickstart: Create a search index in the Azure portal.

Simple query

The following query finds documents whose description fields are most similar to the field of the source document as specified by the `moreLikeThis` parameter:

```
HTTP
```

```
GET /indexes/hotels-sample-index/docs?
moreLikeThis=29&searchFields=Description&api-version=2020-06-30-Preview
```

In this example, the request searches for hotels similar to the one with `HotelId` 29. Rather than using HTTP GET, you can also invoke `MoreLikeThis` using HTTP POST:

HTTP

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30-Preview
    {
      "moreLikeThis": "29",
      "searchFields": "Description"
    }
```

Apply filters

`MoreLikeThis` can be combined with other common query parameters like `$filter`. For instance, the query can be restricted to only hotels whose category is 'Budget' and where the rating is higher than 3.5:

HTTP

```
GET /indexes/hotels-sample-index/docs?
moreLikeThis=20&searchFields=Description&$filter=(Category eq 'Budget' and
Rating gt 3.5)&api-version=2020-06-30-Preview
```

Select fields and limit results

The `$top` selector can be used to limit how many results should be returned in a `MoreLikeThis` query. Also, fields can be selected with `$select`. Here the top three hotels are selected along with their ID, Name, and Rating:

HTTP

```
GET /indexes/hotels-sample-index/docs?
moreLikeThis=20&searchFields=Description&$filter=(Category eq 'Budget' and
Rating gt 3.5)&$top=3&$select=HotelId,HotelName,Rating&api-version=2020-06-
30-Preview
```

Next steps

You can use any web testing tool to experiment with this feature. We recommend using Postman for this exercise.

Explore Azure Cognitive Search REST APIs using Postman

Additional resources

Documentation

How to work with search results - Azure Cognitive Search

Define search result composition, get a document count, sort results, and add content navigation to search results in Azure Cognitive Search.

Create a query - Azure Cognitive Search

Learn how to construct a query request in Cognitive Search, which tools and APIs to use for testing and code, and how query decisions start with index design.

Add custom analyzers to string fields - Azure Cognitive Search

Configure text tokenizers and character filters to perform text analysis on strings during indexing and queries.

Analyze Text (Azure Cognitive Search REST API)

Test predefined and custom analyzers used for breaking text into tokens during Azure Cognitive Search indexing.

C# tutorial indexing multiple Azure data sources - Azure Cognitive Search

Learn how to import data from multiple data sources into a single Azure Cognitive Search index using indexers. This tutorial and sample code are in C#.

Configure relevance scoring - Azure Cognitive Search

Enable Okapi BM25 ranking to upgrade the search ranking and relevance behavior on older Azure Search services.

Suggestions (Azure Cognitive Search REST API)

A query request composed of partial query input, returning matching strings from documents in an Azure Cognitive Search index. Type-ahead query suggestions can populate dropdown search bars or other UI experiences.

Analyzers for linguistic and text processing - Azure Cognitive Search

Assign analyzers to searchable text fields in an index to replace default standard Lucene with custom, predefined or language-specific alternatives.

Show 5 more

Skill context and input annotation language

Article • 02/28/2022 • 5 minutes to read

This article is the reference documentation for skill context and input syntax. It's a full description of the expression language used to construct paths to nodes in an enriched document.

Azure Cognitive Search skills can use and enrich the data coming from the data source and from the output of other skills. The data working set that represents the current state of the indexer work for the current document starts from the raw data coming from the data source and is progressively enriched with each skill iteration's output data. That data is internally organized in a tree-like structure that can be queried to be used as skill inputs or to be added to the index. The nodes in the tree can be simple values such as strings and numbers, arrays, or complex objects and even binary files. Even simple values can be enriched with additional structured information. For example, a string can be annotated with additional information that is stored beneath it in the enrichment tree. The expressions used to query that internal structure use a rich syntax that is detailed in this article. The enriched data structure can be inspected from debug sessions. Expressions querying the structure can also be tested from debug sessions.

Throughout the article, we'll use the following enriched data as an example. This data is typical of the kind of structure you would get when enriching a document using a skillset with OCR, key phrase extraction, text translation, language detection, entity recognition skills and a custom tokenizer skill.

Path	Value
`document`	
`merged_content`	"Study of BMN 110 in Pediatric Patients"...
`keyphrases`	
`[0]`	"Study of BMN"
`[1]`	"Syndrome"
`[2]`	"Pediatric Patients"
...	
`locations`	

Path	Value
[0]	"IVA"
translated_text	"Étude de BMN 110 chez les patients pédiatriques"...
entities	
[0]	
category	"Organization"
subcategory	null
confidenceScore	0.72
length	3
offset	9
text	"BMN"
...	
organizations	
[0]	"BMN"
language	"en"
normalized_images	
[0]	
layoutText	...
text	
words	
[0]	"Study"
[1]	"of"
[2]	"BMN"
[3]	"110"
...	
[1]	

Path	Value
`layoutText`	...
`text`	
`words`	
`[0]`	"it"
`[1]`	"is"
`[2]`	"certainly"
...	
...	
...	

Document root

All the data is under one root element, for which the path is `"/document"`. The root element is the default context for skills.

Simple paths

Simple paths through the internal enriched document can be expressed with simple tokens separated by slashes. This syntax is similar to the JSON Pointer specification.

Object properties

The properties of nodes that represent objects add their values to the tree under the property's name. Those values can be obtained by appending the property name as a token separated by a slash:

Expression	Value
`/document/merged_content/language`	`"en"`

Property name tokens are case-sensitive.

Array item index

Specific elements of an array can be referenced by using their numeric index like a property name:

Expression	Value
`/document/merged_content/keyphrases/1`	`"Syndrome"`
`/document/merged_content/entities/0/text`	`"BMN"`

Escape sequences

There are two characters that have special meaning and need to be escaped if they appear in an expression and must be interpreted as is instead of as their special meaning: `'/'` and `'~'`. Those characters must be escaped respectively as `'~0'` and `'~1'`.

Array enumeration

An array of values can be obtained using the `'*'` token:

Expression	Value
`/document/normalized_images/0/text/words/*`	`["Study", "of", "BMN", "110" ...]`

The `'*'` token doesn't have to be at the end of the path. It's possible to enumerate all nodes matching a path with a star in the middle or with multiple stars:

Expression	Value
`/document/normalized_images/*/text/words/*`	`["Study", "of", "BMN", "110" ... "it", "is", "certainly" ...]`

This example returns a flat list of all matching nodes.

It's possible to maintain more structure and get a separate array for the words of each page by using a `'#'` token instead of the second `'*'` token:

Expression	Value
`/document/normalized_images/*/text/words/#`	`[["Study", "of", "BMN", "110" ...], ["it", "is", "certainly" ...] ...]`

The `'#'` token expresses that the array should be treated as a single value instead of being enumerated.

Enumerating arrays in context

It is often useful to process each element of an array in isolation and have a different set of skill inputs and outputs for each. This can be done by setting the context of the skill to an enumeration instead of the default `"/document"`.

In the following example, we use one of the input expressions we used before, but with a different context that changes the resulting value.

Context	Expression	Values

`/document/normalized_images/*`	`/document/normalized_images/*/text/words/*`	`["Study", "of", "BMN", "110" ...]`
		`["it", "is", "certainly" ...]`
		…

For this combination of context and input, the skill will get executed once for each normalized image: once for `"/document/normalized_images/0"` and once for `"/document/normalized_images/1"`. The two input values corresponding to each skill execution are detailed in the values column.

When enumerating an array in context, any outputs the skill produces will also be added to the document as enrichments of the context. In the above example, an output named `"out"` will have its values for each execution added to the document respectively under `"/document/normalized_images/0/out"` and `"/document/normalized_images/1/out"`.

Literal values

Skill inputs can take literal values as their inputs instead of dynamic values queried from the existing document. This can be achieved by prefixing the value with an equal sign. Values can be numbers, strings or Boolean. String values can be enclosed in single `'` or double `"` quotes.

Expression	Value
`=42`	`42`

Expression	Value
`=2.45E-4`	`0.000245`
`="some string"`	`"some string"`
`='some other string'`	`"some other string"`
`="unicod\u0065"`	`"unicode"`
`=false`	`false`

Composite expressions

It's possible to combine values together using unary, binary and ternary operators. Operators can combine literal values and values resulting from path evaluation. When used inside an expression, paths should be enclosed between `"$("` and `")"`.

Boolean not `'!'`

Expression	Value
`=!false`	`true`

Negative `'-'`

Expression	Value
`=-42`	`-42`
`=-$(/document/merged_content/entities/0/offset)`	`-9`

Addition `'+'`

Expression	Value
`=2+2`	`4`
`=2+$(/document/merged_content/entities/0/offset)`	`11`

Subtraction `'-'`

Expression	Value
`=2-1`	`1`
`=$(/document/merged_content/entities/0/offset)-2`	`7`

Multiplication `'*'`

Expression	Value
`=2*3`	`6`
`=$(/document/merged_content/entities/0/offset)*2`	`18`

Division `'/'`

Expression	Value
`=3/2`	`1.5`
`=$(/document/merged_content/entities/0/offset)/3`	`3`

Modulo `'%'`

Expression	Value
`=15%4`	`3`
`=$(/document/merged_content/entities/0/offset)%2`	`1`

Less than, less than or equal, greater than and greater than or equal `'<'` `'<='` `'>'` `'>='`

Expression	Value
`=15<4`	`false`
`=4<=4`	`true`
`=15>4`	`true`
`=1>=2`	`false`

Equality and non-equality `'=='` `'!='`

Expression	Value
`=15==4`	`false`
`=4==4`	`true`
`=15!=4`	`true`
`=1!=1`	`false`

Logical operations and, or and exclusive or `'&&'` `'||'` `'^'`

Expression	Value
`=true&&true`	`true`
`=true&&false`	`false`
`=true||true`	`true`
`=true||false`	`true`
`=false||false`	`false`
`=true^false`	`true`
`=true^true`	`false`

Ternary operator `'?:'`

It is possible to give an input different values based on the evaluation of a Boolean expression using the ternary operator.

Expression	Value
`=true?"true":"false"`	`"true"`
`=$(/document/merged_content/entities/0/offset)==9?"nine":"not nine"`	`"nine"`

Parentheses and operator priority

Operators are evaluated with priorities that match usual conventions: unary operators, then multiplication, division and modulo, then addition and subtraction, then

comparison, then equality, and then logical operators. Usual associativity rules also apply.

Parentheses can be used to change or disambiguate evaluation order.

Expression	Value
`=3*2+5`	`11`
`=3*(2+5)`	`21`

See also

- Create a skillset in Azure Cognitive Search
- Reference annotations in an Azure Cognitive Search skillset

Built-in skills for text and image processing during indexing (Azure Cognitive Search)

Article • 09/15/2022 • 3 minutes to read

This article describes the skills provided with Azure Cognitive Search that you can include in a skillset to extract content and structure from raw unstructured text and image files. A *skill* is an atomic operation that transforms content in some way. Often, it is an operation that recognizes or extracts text, but it can also be a utility skill that reshapes the enrichments that are already created. Typically, the output is text-based so that it can be used in full text queries.

Built-in skills

Built-in skills are based on pre-trained models from Microsoft, which means you cannot train the model using your own training data. Skills that call the Cognitive Resources APIs have a dependency on those services and are billed at the Cognitive Services pay-as-you-go price when you attach a resource. Other skills are metered by Azure Cognitive Search, or are utility skills that are available at no charge.

The following table enumerates and describes the built-in skills.

OData type	Description	Metered by
Microsoft.Skills.Text.CustomEntityLookupSkill	Looks for text from a custom, user-defined list of words and phrases.	Azure Cognitive Search (pricing)
Microsoft.Skills.Text.KeyPhraseExtractionSkill	This skill uses a pretrained model to detect important phrases based on term placement, linguistic rules, proximity to other terms, and how unusual the term is within the source data.	Cognitive Services (pricing)

OData type	Description	Metered by
Microsoft.Skills.Text.LanguageDetectionSkill	This skill uses a pretrained model to detect which language is used (one language ID per document). When multiple languages are used within the same text segments, the output is the LCID of the predominantly used language.	Cognitive Services (pricing)
Microsoft.Skills.Text.MergeSkill	Consolidates text from a collection of fields into a single field.	Not applicable
Microsoft.Skills.Text.V3.EntityLinkingSkill	This skill uses a pretrained model to generate links for recognized entities to articles in Wikipedia.	Cognitive Services (pricing)
Microsoft.Skills.Text.V3.EntityRecognitionSkill	This skill uses a pretrained model to establish entities for a fixed set of categories: `"Person"`, `"Location"`, `"Organization"`, `"Quantity"`, `"DateTime"`, `"URL"`, `"Email"`, `"PersonType"`, `"Event"`, `"Product"`, `"Skill"`, `"Address"`, `"Phone Number"` and `"IP Address"` fields.	Cognitive Services (pricing)
Microsoft.Skills.Text.PIIDetectionSkill	This skill uses a pretrained model to extract personal information from a given text. The skill also gives various options for masking the detected personal information entities in the text.	Cognitive Services (pricing)
Microsoft.Skills.Text.V3.SentimentSkill	This skill uses a pretrained model to assign sentiment labels (such as "negative", "neutral" and "positive") based on the highest confidence score found by the service at a sentence and document-level on a record by record basis.	Cognitive Services (pricing)
Microsoft.Skills.Text.SplitSkill	Splits text into pages so that you can enrich or augment content incrementally.	Not applicable
Microsoft.Skills.Text.TranslationSkill	This skill uses a pretrained model to translate the input text into a variety of languages for normalization or localization use cases.	Cognitive Services (pricing)

OData type	Description	Metered by
Microsoft.Skills.Vision.ImageAnalysisSkill	This skill uses an image detection algorithm to identify the content of an image and generate a text description.	Cognitive Services (pricing)
Microsoft.Skills.Vision.OcrSkill	Optical character recognition.	Cognitive Services (pricing)
Microsoft.Skills.Util.ConditionalSkill	Allows filtering, assigning a default value, and merging data based on a condition.	Not applicable
Microsoft.Skills.Util.DocumentExtractionSkill	Extracts content from a file within the enrichment pipeline.	Azure Cognitive Search (pricing)
Microsoft.Skills.Util.ShaperSkill	Maps output to a complex type (a multi-part data type, which might be used for a full name, a multi-line address, or a combination of last name and a personal identifier.)	Not applicable

Custom skills

Custom skills are modules that you design, develop, and deploy to the web. You can then call the module from within a skillset as a custom skill.

Type	Description	Metered by
Microsoft.Skills.Custom.WebApiSkill	Allows extensibility of an AI enrichment pipeline by making an HTTP call into a custom Web API	None unless your solution uses a metered Azure service
Microsoft.Skills.Custom.AmlSkill	Allows extensibility of an AI enrichment pipeline with an Azure Machine Learning model	None unless your solution uses a metered Azure service

For guidance on creating a custom skill, see Define a custom interface and Example: Creating a custom skill for AI enrichment.

See also

- How to define a skillset
- Custom Skills interface definition
- Tutorial: Enriched indexing with AI

Conditional cognitive skill

Article • 11/29/2021 • 4 minutes to read

The **Conditional** skill enables Azure Cognitive Search scenarios that require a Boolean operation to determine the data to assign to an output. These scenarios include filtering, assigning a default value, and merging data based on a condition.

The following pseudocode demonstrates what the conditional skill accomplishes:

```
if (condition)
    { output = whenTrue }
else
    { output = whenFalse }
```

> **Note**
>
> This skill isn't bound to Cognitive Services. It is non-billable and has no Cognitive Services key requirement.

@odata.type

Microsoft.Skills.Util.ConditionalSkill

Evaluated fields

This skill is special because its inputs are evaluated fields.

The following items are valid values of an expression:

- Annotation paths (paths in expressions must be delimited by "$(" and ")")
  Examples:

  ```
  "= $(/document)"
  "= $(/document/content)"
  ```

- Literals (strings, numbers, true, false, null)
  Examples:

```
"= 'this is a string'"   // string (note the single quotation marks)
"= 34"                   // number
"= true"                 // Boolean
"= null"                 // null value
```

- Expressions that use comparison operators (==, !=, >=, >, <=, <)
  Examples:

```
"= $(/document/language) == 'en'"
"= $(/document/sentiment) >= 0.5"
```

- Expressions that use Boolean operators (&&, ||, !, ^)
  Examples:

```
"= $(/document/language) == 'en' && $(/document/sentiment) > 0.5"
"= !true"
```

- Expressions that use numeric operators (+, -, *, /, %)
  Examples:

```
"= $(/document/sentiment) + 0.5"         // addition
"= $(/document/totalValue) * 1.10"       // multiplication
"= $(/document/lengthInMeters) / 0.3049" // division
```

Because the conditional skill supports evaluation, you can use it in minor-transformation scenarios. For example, see skill definition 4.

Skill inputs

Inputs are case-sensitive.

Input	Description

Input	Description
condition	This input is an evaluated field that represents the condition to evaluate. This condition should evaluate to a Boolean value (*true* or *false*). Examples: "= true" "= $(/document/language) ==='fr'" "= $(/document/pages/*/language) == $(/document/expectedLanguage)"
whenTrue	This input is an evaluated field that represents the value to return if the condition is evaluated to *true*. Constants strings should be returned in single quotation marks (' and '). Sample values: "= 'contract'" "= $(/document/contractType)" "= $(/document/entities/*)"
whenFalse	This input is an evaluated field that represents the value to return if the condition is evaluated to *false*. Sample values: "= 'contract'" "= $(/document/contractType)" "= $(/document/entities/*)"

Skill outputs

There's a single output that's simply called "output." It returns the value *whenFalse* if the condition is false or *whenTrue* if the condition is true.

Examples

Sample skill definition 1: Filter documents to return only French documents

The following output returns an array of sentences ("/document/frenchSentences") if the language of the document is French. If the language isn't French, the value is set to *null*.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document",
    "inputs": [
        { "name": "condition", "source": "= $(/document/language) == 'fr'"
},
```

```
        { "name": "whenTrue", "source": "/document/sentences" },
        { "name": "whenFalse", "source": "= null" }
    ],
    "outputs": [ { "name": "output", "targetName": "frenchSentences" } ]
}
```

If "/document/frenchSentences" is used as the *context* of another skill, that skill only runs if "/document/frenchSentences" isn't set to *null*.

Sample skill definition 2: Set a default value for a value that doesn't exist

The following output creates an annotation ("/document/languageWithDefault") that's set to the language of the document or to "es" if the language isn't set.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document",
    "inputs": [
        { "name": "condition", "source": "= $(/document/language) == null"
},
        { "name": "whenTrue", "source": "= 'es'" },
        { "name": "whenFalse", "source": "= $(/document/language)" }
    ],
    "outputs": [ { "name": "output", "targetName": "languageWithDefault" } ]
}
```

Sample skill definition 3: Merge values from two fields into one

In this example, some sentences have a *frenchSentiment* property. Whenever the *frenchSentiment* property is null, we want to use the *englishSentiment* value. We assign the output to a member that's called *sentiment* ("/document/sentences/*/sentiment").

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document/sentences/*",
    "inputs": [
        { "name": "condition", "source": "=
$(/document/sentences/*/frenchSentiment) == null" },
        { "name": "whenTrue", "source":
"/document/sentences/*/englishSentiment" },
```

```
        { "name": "whenFalse", "source":
"/document/sentences/*/frenchSentiment" }
    ],
    "outputs": [ { "name": "output", "targetName": "sentiment" } ]
}
```

Transformation example

Sample skill definition 4: Data transformation on a single field

In this example, we receive a *sentiment* that's between 0 and 1. We want to transform it to be between -1 and 1. We can use the conditional skill to do this minor transformation.

In this example, we don't use the conditional aspect of the skill because the condition is always *true*.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ConditionalSkill",
    "context": "/document/sentences/*",
    "inputs": [
        { "name": "condition", "source": "= true" },
        { "name": "whenTrue", "source": "=
$(/document/sentences/*/sentiment) * 2 - 1" },
        { "name": "whenFalse", "source": "= 0" }
    ],
    "outputs": [ { "name": "output", "targetName": "normalizedSentiment" } ]
}
```

Special considerations

Some parameters are evaluated, so you need to be especially careful to follow the documented pattern. Expressions must start with an equals sign. A path must be delimited by "$(" and ")". Make sure to put strings in single quotation marks. That helps the evaluator distinguish between strings and actual paths and operators. Also, make sure to put white space around operators (for instance, a "*" in a path means something different than multiply).

Next steps

- Built-in skills
- How to define a skillset

Additional resources

🕮 Documentation

Text Merge cognitive skill - Azure Cognitive Search

Merge text from a collection of fields into one consolidated field. Use this cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Entity Linking cognitive skill (v3) - Azure Cognitive Search

Extract different linked entities from text in an enrichment pipeline in Azure Cognitive Search.

Shaper cognitive skill - Azure Cognitive Search

Extract metadata and structured information from unstructured data and shape it as a complex type in an AI enrichment pipeline in Azure Cognitive Search.

Create a knowledge store using REST - Azure Cognitive Search

Use the REST API and Postman to create an Azure Cognitive Search knowledge store for persisting AI enrichments from skillset.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Show 5 more

Custom Entity Lookup cognitive skill

Article • 02/27/2023 • 10 minutes to read

The **Custom Entity Lookup** skill is used to detect or recognize entities that you define. During skillset execution, the skill looks for text from a custom, user-defined list of words and phrases. The skill uses this list to label any matching entities found within source documents. The skill also supports a degree of fuzzy matching that can be applied to find matches that are similar but not exact.

> **Note**
>
> This skill isn't bound to a Cognitive Services API but requires a Cognitive Services key to allow more than 20 transactions. This skill is **metered by Cognitive Search**.

@odata.type

Microsoft.Skills.Text.CustomEntityLookupSkill

Data limits

- The maximum input record size supported is 256 MB. If you need to break up your data before sending it to the custom entity lookup skill, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.
- The maximum size of the custom entity definition is 10 MB if it's provided as an external file, specified through the "entitiesDefinitionUri" parameter.
- If the entities are defined inline using the "inlineEntitiesDefinition" parameter, the maximum size is 10 KB.

Skill parameters

Parameters are case-sensitive.

Parameter name	Description

Parameter name	Description
`entitiesDefinitionUri`	Path to an external JSON or CSV file containing all the target text to match against. This entity definition is read at the beginning of an indexer run; any updates to this file mid-run won't be realized until subsequent runs. This file must be accessible over HTTPS. See Custom Entity Definition Format below for expected CSV or JSON schema.
`inlineEntitiesDefinition`	Inline JSON entity definitions. This parameter supersedes the entitiesDefinitionUri parameter if present. No more than 10 KB of configuration may be provided inline. See Custom Entity Definition below for expected JSON schema.
`defaultLanguageCode`	(Optional) Language code of the input text used to tokenize and delineate input text. The following languages are supported: `da, de, en, es, fi, fr, it, pt`. The default is English (`en`). If you pass a `languagecode-countrycode` format, only the `languagecode` part of the format is used.
`globalDefaultCaseSensitive`	(Optional) Default case sensitive value for the skill. If `defaultCaseSensitive` value of an entity isn't specified, this value will become the `defaultCaseSensitive` value for that entity.
`globalDefaultAccentSensitive`	(Optional) Default accent sensitive value for the skill. If `defaultAccentSensitive` value of an entity isn't specified, this value will become the `defaultAccentSensitive` value for that entity.
`globalDefaultFuzzyEditDistance`	(Optional) Default fuzzy edit distance value for the skill. If `defaultFuzzyEditDistance` value of an entity isn't specified, this value will become the `defaultFuzzyEditDistance` value for that entity.

Skill inputs

Input name	Description
`text`	The text to analyze.
`languageCode`	Optional. Default is `"en"`.

Skill outputs

Output name	Description

`entities`	An array of complex types that contains the following fields:
	• `"name"`: The top-level entity; it represents the "normalized" form.
	• `"id"`: A unique identifier for the entity as defined in the "Custom Entity Definition".
	• `"description"`: Entity description as defined by the user in the "Custom Entity Definition Format".
	• `"type"`: Entity type as defined by the user in the "Custom Entity Definition Format".
	• `"subtype"`: Entity subtype as defined by the user in the "Custom Entity Definition Format".
	• `"matches"`: An array of complex types that contain:
	  ○ `"text"` from the source document
	  ○ `"offset"` location where the match was found,
	  ○ `"length"` of the text measured in characters
	  ○ `"matchDistance"` or the number of characters that differ between the match and the entity `"name"`.

Custom entity definition format

There are three approaches for providing the list of custom entities to the Custom Entity Lookup skill:

- .CSV file (UTF-8 encoded)
- .JSON file (UTF-8 encoded)
- Inline within the skill definition

If the definition file is in a .CSV or .JSON file, provide the full path in the "entitiesDefinitionUri" parameter. The file is downloaded at the start of each indexer run. It must remain accessible until the indexer stops.

If you're using an inline definition, specify it under the "inlineEntitiesDefinition" skill parameter.

> **Note**
>
> Indexers support specialized parsing modes for JSON and CSV files. When using the custom entity lookup skill, keep "parsingMode" set to "default". The skill expects JSON and CSV in an unparsed state.

CSV format

You can provide the definition of the custom entities to look for in a Comma-Separated Value (CSV) file by providing the path to the file and setting it in the "entitiesDefinitionUri" skill parameter. The path should be at an https location. The definition file can be up to 10 MB in size.

The CSV format is simple. Each line represents a unique entity, as shown below:

```
Bill Gates, BillG, William H. Gates
Microsoft, MSFT
Satya Nadella
```

In this case, there are three entities that can be returned (Bill Gates, Satya Nadella, Microsoft). Aliases follow after the main entity. A match on an alias is bundled under the primary entity. For example, if the string "William H. Gates" is found in a document, a match for the "Bill Gates" entity will be returned.

JSON format

You can provide the definition of the custom entities to look for in a JSON file as well. The JSON format gives you a bit more flexibility since it allows you to define matching rules per term. For instance, you can specify the fuzzy matching distance (Damerau-Levenshtein distance) for each term or whether the matching should be case-sensitive or not.

Just like with CSV files, you need to provide the path to the JSON file and set it in the "entitiesDefinitionUri" skill parameter. The path should be at an https location. The definition file can be up to 10 MB in size.

The most basic JSON custom entity list definition can be a list of entities to match:

JSON

```
[
    {
        "name" : "Bill Gates"
    },
    {
        "name" : "Microsoft"
    },
    {
        "name" : "Satya Nadella"
```

```
    }
]
```

More complex definitions can provide a user-defined ID, description, type, subtype, and aliases. If an alias term is matched, the entity will be returned as well:

JSON

```
[
    {
        "name" : "Bill Gates",
        "description" : "Microsoft founder." ,
        "aliases" : [
            { "text" : "William H. Gates", "caseSensitive" : false },
            { "text" : "BillG", "caseSensitive" : true }
        ]
    },
    {
        "name" : "Xbox One",
        "type": "Hardware",
        "subtype" : "Gaming Device",
        "id" : "4e36bf9d-5550-4396-8647-8e43d7564a76",
        "description" : "The Xbox One product"
    },
    {
        "name" : "LinkedIn" ,
        "description" : "The LinkedIn company",
        "id" : "differentIdentifyingScheme123",
        "fuzzyEditDistance" : 0
    },
    {
        "name" : "Microsoft" ,
        "description" : "Microsoft Corporation",
        "id" : "differentIdentifyingScheme987",
        "defaultCaseSensitive" : false,
        "defaultFuzzyEditDistance" : 1,
        "aliases" : [
            { "text" : "MSFT", "caseSensitive" : true }
        ]
    }
]
```

The tables below describe the configuration parameters you can set when defining custom entities:

Field name	Description
`name`	The top-level entity descriptor. Matches in the skill output will be grouped by this name, and it should represent the "normalized" form of the text being found.

Field name	Description
`description`	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
`type`	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
`subtype`	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
`id`	(Optional) This field can be used as a passthrough for custom metadata about the matched text(s). The value of this field will appear with every match of its entity in the skill output.
`caseSensitive`	(Optional) Defaults to false. Boolean value denoting whether comparisons with the entity name should be sensitive to character casing. Sample case insensitive matches of "Microsoft" could be: microsoft, microSoft, MICROSOFT
`accentSensitive`	(Optional) Defaults to false. Boolean value denoting whether accented and unaccented letters such as 'é' and 'e' should be identical.
`fuzzyEditDistance`	(Optional) Defaults to 0. Maximum value of 5. Denotes the acceptable number of divergent characters that would still constitute a match with the entity name. The smallest possible fuzziness for any given match is returned. For instance, if the edit distance is set to 3, "Windows 10" would still match "Windows", "Windows10" and "windows 7". When case sensitivity is set to false, case differences do NOT count towards fuzziness tolerance, but otherwise do.
`defaultCaseSensitive`	(Optional) Changes the default case sensitivity value for this entity. It can be used to change the default value of all aliases caseSensitive values.
`defaultAccentSensitive`	(Optional) Changes the default accent sensitivity value for this entity. It can be used to change the default value of all aliases accentSensitive values.
`defaultFuzzyEditDistance`	(Optional) Changes the default fuzzy edit distance value for this entity. It can be used to change the default value of all aliases fuzzyEditDistance values.
`aliases`	(Optional) An array of complex objects that can be used to specify alternative spellings or synonyms to the root entity name.

Alias properties	Description
`text`	The alternative spelling or representation of some target entity name.
`caseSensitive`	(Optional) Acts the same as root entity "caseSensitive" parameter above, but applies to only this one alias.
`accentSensitive`	(Optional) Acts the same as root entity "accentSensitive" parameter above, but applies to only this one alias.
`fuzzyEditDistance`	(Optional) Acts the same as root entity "fuzzyEditDistance" parameter above, but applies to only this one alias.

Inline format

In some cases, it may be more convenient to embed the custom entity definition so that its inline with the skill definition. You can use the same JSON format as the one described above, except that it's included within the skill definition. Only configurations that are less than 10 KB in size (serialized size) can be defined inline.

Sample skill definition

A sample skill definition using an inline format is shown below:

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.CustomEntityLookupSkill",
  "context": "/document",
  "inlineEntitiesDefinition":
  [
    {
      "name" : "Bill Gates",
      "description" : "Microsoft founder." ,
      "aliases" : [
          { "text" : "William H. Gates", "caseSensitive" : false },
          { "text" : "BillG", "caseSensitive" : true }
      ]
    },
    {
      "name" : "Xbox One",
      "type": "Hardware",
      "subtype" : "Gaming Device",
      "id" : "4e36bf9d-5550-4396-8647-8e43d7564a76",
      "description" : "The Xbox One product"
    }
  ],
  "inputs": [
```

```
      {
        "name": "text",
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "entities",
        "targetName": "matchedEntities"
      }
    ]
  }
```

Alternatively, you can point to an external entities definition file. A sample skill definition using the `entitiesDefinitionUri` format is shown below:

JSON

```
  {
    "@odata.type": "#Microsoft.Skills.Text.CustomEntityLookupSkill",
    "context": "/document",
    "entitiesDefinitionUri": "https://myblobhost.net/keyWordsConfig.csv",
    "inputs": [
      {
        "name": "text",
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "entities",
        "targetName": "matchedEntities"
      }
    ]
  }
```

Sample index definition

This section provides a sample index definition. Both "entities" and "matches" are arrays of complex types. You can have multiple entities per document, and multiple matches for each entity.

JSON

```
{
  "name": "entities",
  "type": "Collection(Edm.ComplexType)",
  "fields": [
    {
```

```
      "name": "name",
      "type": "Edm.String",
      "facetable": false,
      "filterable": false,
      "retrievable": true,
      "searchable": true,
      "sortable": false,
    },
    {
      "name": "id",
      "type": "Edm.String",
      "facetable": false,
      "filterable": false,
      "retrievable": true,
      "searchable": false,
      "sortable": false,
    },
    {
      "name": "description",
      "type": "Edm.String",
      "facetable": false,
      "filterable": false,
      "retrievable": true,
      "searchable": true,
      "sortable": false,
    },
    {
      "name": "type",
      "type": "Edm.String",
      "facetable": true,
      "filterable": true,
      "retrievable": true,
      "searchable": false,
      "sortable": false,
    },
    {
      "name": "subtype",
      "type": "Edm.String",
      "facetable": true,
      "filterable": true,
      "retrievable": true,
      "searchable": false,
      "sortable": false,
    },
    {
      "name": "matches",
      "type": "Collection(Edm.ComplexType)",
      "fields": [
        {
          "name": "text",
          "type": "Edm.String",
          "facetable": false,
          "filterable": false,
          "retrievable": true,
          "searchable": true,
```

```
          "sortable": false,
        },
        {
          "name": "offset",
          "type": "Edm.Int32",
          "facetable": true,
          "filterable": true,
          "retrievable": true,
          "sortable": false,
        },
        {
          "name": "length",
          "type": "Edm.Int32",
          "facetable": true,
          "filterable": true,
          "retrievable": true,
          "sortable": false,
        },
        {
          "name": "matchDistance",
          "type": "Edm.Double",
          "facetable": true,
          "filterable": true,
          "retrievable": true,
          "sortable": false,
        }
      ]
    }
  ]
}
```

Sample input data

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "The company, Microsoft, was founded by Bill Gates.
Microsoft's gaming console is called Xbox",
             "languageCode": "en"
           }
      }
    ]
}
```

Sample output

JSON

```
  {
    "values" :
    [
      {
        "recordId": "1",
        "data" : {
          "entities": [
            {
              "name" : "Microsoft",
              "description" : "This document refers to Microsoft the
company",
              "id" : "differentIdentifyingScheme987",
              "matches" : [
                {
                  "text" : "microsoft",
                  "offset" : 13,
                  "length" : 9,
                  "matchDistance" : 0
                },
                {
                  "text" : "Microsoft",
                  "offset" : 49,
                  "length" : 9,
                  "matchDistance" : 0
                }
              ]
            },
            {
              "name" : "Bill Gates",
              "description" : "William Henry Gates III, founder of
Microsoft.",
              "matches" : [
                {
                  "text" : "Bill Gates",
                  "offset" : 37,
                  "length" : 10,
                  "matchDistance" : 0
                }
              ]
            }
          ]
        }
      }
    ]
  }
```

Warnings

`"Reached maximum capacity for matches, skipping all further duplicate matches."`

This warning will be emitted if the number of matches detected is greater than the maximum allowed. No more duplicate matches will be returned. If you need a higher threshold, you can file a support ticket for assistance with your individual use case.

See also

- Custom Entity Lookup sample and readme
- Built-in skills
- How to define a skillset
- Entity Recognition skill (to search for well known entities)

Additional resources

Documentation

Map enrichments in indexers - Azure Cognitive Search

Export the enriched content created by a skillset by mapping its output fields to fields in a search index.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Configure semantic search - Azure Cognitive Search

Set a semantic query type to attach the deep learning models of semantic search.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Create a skillset - Azure Cognitive Search

A skillset defines data extraction, natural language processing, and image analysis steps. A skillset is attached to indexer. It's used to enrich and extract information from source data for use in Azure Cognitive Search.

Return a semantic answer - Azure Cognitive Search

Describes the composition of a semantic answer and how to obtain answers from a result set.

Show 5 more

Document Extraction cognitive skill

Article • 05/10/2022 • 4 minutes to read

The **Document Extraction** skill extracts content from a file within the enrichment pipeline. This allows you to take advantage of the document extraction step that normally happens before the skillset execution with files that may be generated by other skills.

> **Note**
>
> This skill isn't bound to Cognitive Services and has no Cognitive Services key requirement. This skill extracts text and images. Text extraction is free. Image extraction is **metered by Azure Cognitive Search**. On a free search service, the cost of 20 transactions per indexer per day is absorbed so that you can complete quickstarts, tutorials, and small projects at no charge. For Basic, Standard, and above, image extraction is billable.

@odata.type

Microsoft.Skills.Util.DocumentExtractionSkill

Supported document formats

The DocumentExtractionSkill can extract text from the following document formats:

- CSV (see Indexing CSV blobs)
- EML
- EPUB
- GZ
- HTML
- JSON (see Indexing JSON blobs)
- KML (XML for geographic representations)
- Microsoft Office formats: DOCX/DOC/DOCM, XLSX/XLS/XLSM, PPTX/PPT/PPTM, MSG (Outlook emails), XML (both 2003 and 2006 WORD XML)
- Open Document formats: ODT, ODS, ODP
- PDF
- Plain text files (see also Indexing plain text)
- RTF
- XML

- ZIP

Skill parameters

Parameters are case-sensitive.

Inputs	Allowed Values	Description

`parsingMode`	`default`
	`text`
	`json`
		Set to `default` for document extraction from files that are not pure text or json. For source files that contain mark up (such as PDF, HTML, RTF, and Microsoft Office files), use the default to extract just the text, minus any markup language or tags. If `parsingMode` is not defined explicitly, it will be set to `default`.
		
		Set to `text` if source files are TXT. This parsing mode improves performance on plain text files. If files include markup, this mode will preserve the tags in the final output.
		
		Set to `json` to extract structured content from json files.

`dataToExtract`	`contentAndMetadata` `allMetadata`	Set to `contentAndMetadata` to extract all metadata and textual content from each file. If `dataToExtract` is not defined explicitly, it will be set to `contentAndMetadata`. Set to `allMetadata` to extract only the metadata properties for the content type (for example, metadata unique to just .png files).
`configuration`	See below.	A dictionary of optional parameters that adjust how the document extraction is performed. See the below table for descriptions of supported configuration properties.

Configuration Parameter	Allowed Values	Description

Configuration Parameter	Allowed Values	Description

`imageAction`	`none`
	`generateNormalizedImages`
	`generateNormalizedImagePerPage`
		Set to `none` to ignore embedded images or image files in the data set, or if the source data does not include image files. This is the default.
		
		For OCR and image analysis, set to `generateNormalizedImages` to have the skill create an array of normalized images as part of document cracking. This action requires that `parsingMode` is set to `default` and `dataToExtract` is set to `contentAndMetadata`. A normalized image refers to additional processing resulting in uniform image output, sized and rotated to promote consistent rendering when you include images in visual search results (for example, same-size photographs in a graph control as seen in the JFK demo). This information is generated for each image when you use this option.
		
		If you set to `generateNormalizedImagePerPage`, PDF files will be treated differently in that instead of extracting embedded images, each page will be rendered as an image and normalized accordingly. Non-PDF file types will be treated the same as if `generateNormalizedImages` was set.

`normalizedImageMaxWidth`	Any integer between 50-10000	The maximum width (in pixels) for normalized images generated. The default is 2000.
`normalizedImageMaxHeight`	Any integer between 50-10000	The maximum height (in pixels) for normalized images generated. The default is 2000.

> **Note**
>
> The default of 2000 pixels for the normalized images maximum width and height is based on the maximum sizes supported by the **OCR skill** and the **image analysis skill**. The **OCR skill** supports a maximum width and height of 4200 for non-English languages, and 10000 for English. If you increase the maximum limits, processing could fail on larger images depending on your skillset definition and the language of the documents.

Skill inputs

Input name	Description
`file_data`	The file that content should be extracted from.

The "file_data" input must be an object defined as:

JSON

```
{
  "$type": "file",
  "data": "BASE64 encoded string of the file"
}
```

Alternatively, it can be defined as:

JSON

```
{
  "$type": "file",
  "url": "URL to download file",
  "sasToken": "OPTIONAL: SAS token for authentication if the URL provided is
for a file in blob storage"
}
```

The file reference object can be generated one of three ways:

- Setting the `allowSkillsetToReadFileData` parameter on your indexer definition to "true". This will create a path `/document/file_data` that is an object representing the original file data downloaded from your blob data source. This parameter only applies to files in Blob storage.

- Setting the `imageAction` parameter on your indexer definition to a value other than `none`. This creates an array of images that follows the required convention for input to this skill if passed individually (that is, `/document/normalized_images/*`).
- Having a custom skill return a json object defined EXACTLY as above. The `$type` parameter must be set to exactly `file` and the `data` parameter must be the base 64 encoded byte array data of the file content, or the `url` parameter must be a correctly formatted URL with access to download the file at that location.

Skill outputs

Output name	Description
`content`	The textual content of the document.
`normalized_images`	When the `imageAction` is set to a value other than `none`, the new *normalized_images* field will contain an array of images. See Extract text and information from images for more details on the output format.

Sample definition

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.DocumentExtractionSkill",
    "parsingMode": "default",
    "dataToExtract": "contentAndMetadata",
    "configuration": {
        "imageAction": "generateNormalizedImages",
        "normalizedImageMaxWidth": 2000,
        "normalizedImageMaxHeight": 2000
    },
    "context": "/document",
    "inputs": [
      {
        "name": "file_data",
        "source": "/document/file_data"
      }
    ],
    "outputs": [
      {
        "name": "content",
        "targetName": "extracted_content"
      },
      {
        "name": "normalized_images",
        "targetName": "extracted_normalized_images"
```

```
      }
    ]
  }
```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data":
      {
        "file_data": {
          "$type": "file",
          "data": "aGVsbG8="
        }
      }
    }
  ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "content": "hello",
        "normalized_images": []
      }
    }
  ]
}
```

See also

- Built-in skills
- How to define a skillset
- How to process and extract information from images in cognitive search scenarios

Entity Linking cognitive skill (v3)

Article • 03/14/2023 • 2 minutes to read

The **Entity Linking** skill (v3) returns a list of recognized entities with links to articles in a well-known knowledge base (Wikipedia).

> **Note**
>
> This skill is bound to the **Entity Linking** machine learning models in **Azure Cognitive Services for Language** and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.V3.EntityLinkingSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the EntityLinking skill, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameter names are case-sensitive and are all optional.

Parameter name	Description
`defaultLanguageCode`	Language code of the input text. If the default language code is not specified, English (en) will be used as the default language code. See the full list of supported languages.
`minimumPrecision`	A value between 0 and 1. If the confidence score (in the `entities` output) is lower than this value, the entity is not returned. The default is 0.
`modelVersion`	(Optional) Specifies the version of the model to use when calling entity linking. It will default to the latest available when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

Input name	Description
`languageCode`	A string indicating the language of the records. If this parameter is not specified, the default language code will be used to analyze the records. See the full list of supported languages.
`text`	The text to analyze.

Skill outputs

Output name	Description
`entities`	An array of complex types that contains the following fields: • `"name"` (The actual entity name as it appears in the text) • `"id"` • `"language"` (The language of the text as determined by the skill) • `"url"` (The linked url to this entity) • "`bingId`" (The bingId for this linked entity) • `"dataSource"` (The data source associated with the url) • `"matches"` (An array of complex types that contains: `text`, `offset`, `length` and `confidenceScore`)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityLinkingSkill",
  "context": "/document",
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.5,
  "inputs": [
      {
          "name": "text",
          "source": "/document/content"
      },
      {
          "name": "languageCode",
          "source": "/document/language"
      }
  ],
  "outputs": [
      {
```

```
            "name": "entities",
            "targetName": "entities"
        }
    ]
}
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "Microsoft is liked by many.",
             "languageCode": "en"
           }
      }
    ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data" :
      {
        "entities": [
          {
            "name": "Microsoft",
            "id": "Microsoft",
            "language": "en",
            "url": "https://en.wikipedia.org/wiki/Microsoft",
            "bingId": "a093e9b9-90f5-a3d5-c4b8-5855e1b01f85",
            "dataSource": "Wikipedia",
            "matches": [
                {
                    "text": "Microsoft",
                    "offset": 0,
                    "length": 9,
                    "confidenceScore": 0.13
                }
            ]
```

```
            }
          ],
        }
      }
    ]
  }
```

The offsets returned for entities in the output of this skill are directly returned from the Language Service APIs, which means if you are using them to index into the original string, you should use the StringInfo class in .NET in order to extract the correct content. For more information, see Multilingual and emoji support in Language service features.

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- Built-in skills
- How to define a skillset

Additional resources

🕮 Documentation

Text Merge cognitive skill - Azure Cognitive Search

Merge text from a collection of fields into one consolidated field. Use this cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Create Skillset (Azure Cognitive Search REST API)

A skillset is a collection of cognitive skills that comprise an enriched indexing pipeline in Azure Cognitive Search.

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Show 5 more

Entity Recognition cognitive skill (v3)

Article • 03/14/2023 • 4 minutes to read

The **Entity Recognition** skill (v3) extracts entities of different types from text. These entities fall under 14 distinct categories, ranging from people and organizations to URLs and phone numbers. This skill uses the Named Entity Recognition machine learning models provided by Azure Cognitive Services for Language.

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.V3.EntityRecognitionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the EntityRecognition skill, consider using the Text Split skill. When using a split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive and are all optional.

Parameter name	Description
`categories`	Array of categories that should be extracted. Possible category types: `"Person"`, `"Location"`, `"Organization"`, `"Quantity"`, `"DateTime"`, `"URL"`, `"Email"`, `"personType"`, `"Event"`, `"Product"`, `"Skill"`, `"Address"`, `"phoneNumber"`, `"ipAddress"`. If no category is provided, all types are returned.

Parameter name	Description
`defaultLanguageCode`	Language code of the input text. If the default language code is not specified, English (en) will be used as the default language code. See the full list of supported languages. Not all entity categories are supported for all languages; see note below.
`minimumPrecision`	A value between 0 and 1. If the confidence score (in the `namedEntities` output) is lower than this value, the entity is not returned. The default is 0.
`modelVersion`	(Optional) Specifies the version of the model to use when calling the entity recognition API. It will default to the latest available when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

Input name	Description
`languageCode`	A string indicating the language of the records. If this parameter is not specified, the default language code will be used to analyze the records. See the full list of supported languages.
`text`	The text to analyze.

Skill outputs

> **Note**
>
> Not all entity categories are supported for all languages. See **Supported Named Entity Recognition (NER) entity categories** to know which entity categories are supported for the language you will be using.

Output name	Description
`persons`	An array of strings where each string represents the name of a person.
`locations`	An array of strings where each string represents a location.
`organizations`	An array of strings where each string represents an organization.
`quantities`	An array of strings where each string represents a quantity.

Output name	Description
`dateTimes`	An array of strings where each string represents a DateTime (as it appears in the text) value.
`urls`	An array of strings where each string represents a URL
`emails`	An array of strings where each string represents an email
`personTypes`	An array of strings where each string represents a PersonType
`events`	An array of strings where each string represents an event
`products`	An array of strings where each string represents a product
`skills`	An array of strings where each string represents a skill
`addresses`	An array of strings where each string represents an address
`phoneNumbers`	An array of strings where each string represents a telephone number
`ipAddresses`	An array of strings where each string represents an IP Address
`namedEntities`	An array of complex types that contains the following fields: • category • subcategory • confidenceScore (Higher value means it's more to be a real entity) • length (The length(number of characters) of this entity) • offset (The location where it was found in the text) • text (The actual entity name as it appears in the text)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
  "context": "/document",
  "categories": [ "Person", "Email"],
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.5,
  "inputs": [
      {
          "name": "text",
          "source": "/document/content"
      },
      {
          "name": "languageCode",
          "source": "/document/language"
```

```
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        },
        {
            "name": "emails",
            "targetName": "emails"
        },
        {
            "name": "namedEntities",
            "targetName": "namedEntities"
        }
    ]
  }
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "Contoso Corporation was founded by Jean Martin. They
can be reached at contact@contoso.com",
             "languageCode": "en"
           }
      }
    ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data" :
      {
        "people": [ "Jean Martin"],
        "emails":["contact@contoso.com"],
        "namedEntities":
```

```
        [
          {
            "category": "Person",
            "subcategory": null,
            "length": 11,
            "offset": 35,
            "confidenceScore": 0.98,
            "text": "Jean Martin"
          },
          {
            "category": "Email",
            "subcategory": null,
            "length": 19,
            "offset": 71,
            "confidenceScore": 0.8,
            "text": "contact@contoso.com"
          }
        ],
      }
    }
  ]
}
```

The offsets returned for entities in the output of this skill are directly returned from the Language Service APIs, which means if you are using them to index into the original string, you should use the StringInfo class in .NET in order to extract the correct content. For more information, see Multilingual and emoji support in Language service features.

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- Built-in skills
- How to define a skillset

Image Analysis cognitive skill

Article • 07/06/2022 • 9 minutes to read

The **Image Analysis** skill extracts a rich set of visual features based on the image content. For example, you can generate a caption from an image, generate tags, or identify celebrities and landmarks. This article is the reference documentation for the **Image Analysis** skill. See Extract text and information from images for usage instructions.

This skill uses the machine learning models provided by Computer Vision in Cognitive Services. **Image Analysis** works on images that meet the following requirements:

- The image must be presented in JPEG, PNG, GIF or BMP format
- The file size of the image must be less than 4 megabytes (MB)
- The dimensions of the image must be greater than 50 x 50 pixels

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.
>
> In addition, image extraction is **billable by Azure Cognitive Search**.

@odata.type

Microsoft.Skills.Vision.ImageAnalysisSkill

Skill parameters

Parameters are case-sensitive.

Parameter name	Description
`defaultLanguageCode`	A string indicating the language to return. The service returns recognition results in a specified language. If this parameter isn't specified, the default value is "en". Supported languages include all of the generally available languages of Cognitive Services Computer Vision.

Parameter name	Description
`visualFeatures`	An array of strings indicating the visual feature types to return. Valid visual feature types include: • *adult* - detects if the image is pornographic (depicts nudity or a sex act), gory (depicts extreme violence or blood) or suggestive (also known as racy content). • *brands* - detects various brands within an image, including the approximate location. • *categories* - categorizes image content according to a taxonomy defined by Cognitive Services. • *description* - describes the image content with a complete sentence in supported languages. • *faces* - detects if faces are present. If present, generates coordinates, gender and age. • *objects* - detects various objects within an image, including the approximate location. • *tags* - tags the image with a detailed list of words related to the image content. Names of visual features are case-sensitive. Both *color* and *imageType* visual features have been deprecated, but you can access this functionality through a custom skill. Refer to the Computer Vision Image Analysis documentation on which visual features are supported with each `defaultLanguageCode`.
`details`	An array of strings indicating which domain-specific details to return. Valid visual feature types include: • *celebrities* - identifies celebrities if detected in the image. • *landmarks* - identifies landmarks if detected in the image.

Skill inputs

Input name	Description
`image`	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure Blob indexer when `imageAction` is set to a value other than `none`.

Skill outputs

Output name	Description

Output name	Description
`adult`	Output is a single adult object of a complex type, consisting of boolean fields (`isAdultContent`, `isGoryContent`, `isRacyContent`) and double type scores (`adultScore`, `goreScore`, `racyScore`).
`brands`	Output is an array of brand objects, where the object is a complex type consisting of `name` (string) and a `confidence` score (double). It also returns a `rectangle` with four bounding box coordinates (`x`, `y`, `w`, `h`, in pixels) indicating placement inside the image. For the rectangle, `x` and `y` are the top left. Bottom left is `x`, `y+h`. Top right is `x+w`, `y`. Bottom right is `x+w`, `y+h`.
`categories`	Output is an array of category objects, where each category object is a complex type consisting of a `name` (string), `score` (double), and optional `detail` that contains celebrity or landmark details. See the category taxonomy for the full list of category names. A detail is a nested complex type. A celebrity detail consists of a name, confidence score, and face bounding box. A landmark detail consists of a name and confidence score.
`description`	Output is a single description object of a complex type, consisting of lists of `tags` and `caption` (an array consisting of `Text` (string) and `confidence` (double)).
`faces`	Complex type consisting of `age`, `gender`, and `faceBoundingBox` having four bounding box coordinates (in pixels) indicating placement inside the image. Coordinates are `top`, `left`, `width`, `height`.
`objects`	Output is an array of visual feature objects. Each object is a complex type, consisting of `object` (string), `confidence` (double), `rectangle` (with four bounding box coordinates indicating placement inside the image), and a `parent` that contains an object name and confidence .
`tags`	Output is an array of imageTag objects, where a tag object is a complex type consisting of `name` (string), `hint` (string), and `confidence` (double). The addition of a hint is rare. It's only generated if a tag is ambiguous. For example, an image tagged as "curling" might have a hint of "sports" to better indicate its content.

Sample skill definition

JSON

```
{
    "description": "Extract image analysis.",
    "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
    "context": "/document/normalized_images/*",
    "defaultLanguageCode": "en",
    "visualFeatures": [
        "adult",
        "brands",
```

```
        "categories",
        "description",
        "faces",
        "objects",
        "tags"
    ],
    "inputs": [
        {
            "name": "image",
            "source": "/document/normalized_images/*"
        }
    ],
    "outputs": [
        {
            "name": "adult"
        },
        {
            "name": "brands"
        },
        {
            "name": "categories"
        },
        {
            "name": "description"
        },
        {
            "name": "faces"
        },
        {
            "name": "objects"
        },
        {
            "name": "tags"
        }
    ]
}
```

Sample index

For single objects (such as `adult` and `description`), you can structure them in the index as a `Collection(Edm.ComplexType)` to return `adult` and `description` output for all of them. For more information about mapping outputs to index fields, see Flattening information from complex types.

JSON

```
{
    "fields": [
        {
            "name": "metadata_storage_name",
            "type": "Edm.String",
```

```
            "key": true,
            "searchable": true,
            "filterable": false,
            "facetable": false,
            "sortable": true
        },
        {
            "name": "metadata_storage_path",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "facetable": false,
            "sortable": true
        },
        {
            "name": "content",
            "type": "Edm.String",
            "sortable": false,
            "searchable": true,
            "filterable": false,
            "facetable": false
        },
        {
            "name": "adult",
            "type": "Edm.ComplexType",
            "fields": [
                {
                    "name": "isAdultContent",
                    "type": "Edm.Boolean",
                    "searchable": false,
                    "filterable": true,
                    "facetable": true
                },
                {
                    "name": "isGoryContent",
                    "type": "Edm.Boolean",
                    "searchable": false,
                    "filterable": true,
                    "facetable": true
                },
                {
                    "name": "isRacyContent",
                    "type": "Edm.Boolean",
                    "searchable": false,
                    "filterable": true,
                    "facetable": true
                },
                {
                    "name": "adultScore",
                    "type": "Edm.Double",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                },
                {
```

```
                "name": "goreScore",
                "type": "Edm.Double",
                "searchable": false,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "racyScore",
                "type": "Edm.Double",
                "searchable": false,
                "filterable": false,
                "facetable": false
            }
        ]
    },
    {
        "name": "brands",
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "name",
                "type": "Edm.String",
                "searchable": true,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "confidence",
                "type": "Edm.Double",
                "searchable": false,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "rectangle",
                "type": "Edm.ComplexType",
                "fields": [
                    {
                        "name": "x",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "y",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "w",
                        "type": "Edm.Int32",
                        "searchable": false,
```

```
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "h",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    }
                ]
            }
        ]
    },
    {
        "name": "categories",
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "name",
                "type": "Edm.String",
                "searchable": true,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "score",
                "type": "Edm.Double",
                "searchable": false,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "detail",
                "type": "Edm.ComplexType",
                "fields": [
                    {
                        "name": "celebrities",
                        "type": "Collection(Edm.ComplexType)",
                        "fields": [
                            {
                                "name": "name",
                                "type": "Edm.String",
                                "searchable": true,
                                "filterable": false,
                                "facetable": false
                            },
                            {
                                "name": "faceBoundingBox",
                                "type": "Collection(Edm.ComplexType)",
                                "fields": [
                                    {
                                        "name": "x",
                                        "type": "Edm.Int32",
                                        "searchable": false,
```

```
                                        "filterable": false,
                                        "facetable": false
                                    },
                                    {
                                        "name": "y",
                                        "type": "Edm.Int32",
                                        "searchable": false,
                                        "filterable": false,
                                        "facetable": false
                                    }
                                ]
                            },
                            {
                                "name": "confidence",
                                "type": "Edm.Double",
                                "searchable": false,
                                "filterable": false,
                                "facetable": false
                            }
                        ]
                    },
                    {
                        "name": "landmarks",
                        "type": "Collection(Edm.ComplexType)",
                        "fields": [
                            {
                                "name": "name",
                                "type": "Edm.String",
                                "searchable": true,
                                "filterable": false,
                                "facetable": false
                            },
                            {
                                "name": "confidence",
                                "type": "Edm.Double",
                                "searchable": false,
                                "filterable": false,
                                "facetable": false
                            }
                        ]
                    }
                ]
            }
        ]
    },
    {
        "name": "description",
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "tags",
                "type": "Collection(Edm.String)",
                "searchable": true,
                "filterable": false,
                "facetable": false
```

```
                },
                {
                    "name": "captions",
                    "type": "Collection(Edm.ComplexType)",
                    "fields": [
                        {
                            "name": "text",
                            "type": "Edm.String",
                            "searchable": true,
                            "filterable": false,
                            "facetable": false
                        },
                        {
                            "name": "confidence",
                            "type": "Edm.Double",
                            "searchable": false,
                            "filterable": false,
                            "facetable": false
                        }
                    ]
                }
            ]
        },
        {
            "name": "faces",
            "type": "Collection(Edm.ComplexType)",
            "fields": [
                {
                    "name": "age",
                    "type": "Edm.Int32",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                },
                {
                    "name": "gender",
                    "type": "Edm.String",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                },
                {
                    "name": "faceBoundingBox",
                    "type": "Collection(Edm.ComplexType)",
                    "fields": [
                        {
                            "name": "top",
                            "type": "Edm.Int32",
                            "searchable": false,
                            "filterable": false,
                            "facetable": false
                        },
                        {
                            "name": "left",
                            "type": "Edm.Int32",
```

```
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "width",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "height",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    }
                ]
            }
        ]
    },
    {
        "name": "objects",
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "object",
                "type": "Edm.String",
                "searchable": true,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "confidence",
                "type": "Edm.Double",
                "searchable": false,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "rectangle",
                "type": "Edm.ComplexType",
                "fields": [
                    {
                        "name": "x",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "y",
                        "type": "Edm.Int32",
                        "searchable": false,
```

```
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "w",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "h",
                        "type": "Edm.Int32",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    }
                ]
            },
            {
                "name": "parent",
                "type": "Edm.ComplexType",
                "fields": [
                    {
                        "name": "object",
                        "type": "Edm.String",
                        "searchable": true,
                        "filterable": false,
                        "facetable": false
                    },
                    {
                        "name": "confidence",
                        "type": "Edm.Double",
                        "searchable": false,
                        "filterable": false,
                        "facetable": false
                    }
                ]
            }
        ]
    },
    {
        "name": "tags",
        "type": "Collection(Edm.ComplexType)",
        "fields": [
            {
                "name": "name",
                "type": "Edm.String",
                "searchable": true,
                "filterable": false,
                "facetable": false
            },
            {
                "name": "hint",
                "type": "Edm.String",
```

```
                    "searchable": true,
                    "filterable": false,
                    "facetable": false
                },
                {
                    "name": "confidence",
                    "type": "Edm.Double",
                    "searchable": false,
                    "filterable": false,
                    "facetable": false
                }
            ]
        }
    ]
}
```

Sample output field mapping

The target field can be a complex field or collection. The index definition specifies any subfields.

JSON

```
"outputFieldMappings": [
    {
        "sourceFieldName": "/document/normalized_images/*/adult",
        "targetFieldName": "adult"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/brands/*",
        "targetFieldName": "brands"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/categories/*",
        "targetFieldName": "categories"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/description",
        "targetFieldName": "description"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/faces/*",
        "targetFieldName": "faces"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/objects/*",
        "targetFieldName": "objects"
    },
    {
        "sourceFieldName": "/document/normalized_images/*/tags/*",
```

```
        "targetFieldName": "tags"
    }
```

Variation on output field mappings (nested properties)

You can define output field mappings to lower-level properties, such as just celebrities or landmarks. In this case, make sure your index schema has a field to contain each detail specifically.

JSON

```
"outputFieldMappings": [
    {
        "sourceFieldName":
"/document/normalized_images/*/categories/detail/celebrities/*",
        "targetFieldName": "celebrities"
    },
    {
        "sourceFieldName":
"/document/normalized_images/*/categories/detail/landmarks/*",
        "targetFieldName": "landmarks"
    }
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "image": {
                    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",
                    "width": 500,
                    "height": 300,
                    "originalWidth": 5000,
                    "originalHeight": 3000,
                    "rotationFromOriginal": 90,
                    "contentOffset": 500,
                    "pageNumber": 2
                }
            }
        }
    ]
}
```

Sample output

JSON


```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "categories": [
          {
            "name": "abstract_",
            "score": 0.00390625
          },
          {
            "name": "people_",
            "score": 0.83984375,
            "detail": {
              "celebrities": [
                {
                  "name": "Satya Nadella",
                  "faceBoundingBox": [
                        {
                            "x": 273,
                            "y": 309
                        },
                        {
                            "x": 395,
                            "y": 309
                        },
                        {
                            "x": 395,
                            "y": 431
                        },
                        {
                            "x": 273,
                            "y": 431
                        }
                    ],
                  "confidence": 0.999028444
                }
              ],
              "landmarks": [ ]
            }
          }
        ],
        "adult": {
          "isAdultContent": false,
          "isRacyContent": false,
          "isGoryContent": false,
          "adultScore": 0.0934349000453949,
          "racyScore": 0.068613491952419281,
          "goreScore": 0.08928389008070282
```

```
    },
    "tags": [
      {
        "name": "person",
        "confidence": 0.98979085683822632
      },
      {
        "name": "man",
        "confidence": 0.94493889808654785
      },
      {
        "name": "outdoor",
        "confidence": 0.938492476940155
      },
      {
        "name": "window",
        "confidence": 0.89513939619064331
      }
    ],
    "description": {
      "tags": [
        "person",
        "man",
        "outdoor",
        "window",
        "glasses"
      ],
      "captions": [
        {
          "text": "Satya Nadella sitting on a bench",
          "confidence": 0.48293603002174407
        }
      ]
    },
    "faces": [
      {
        "age": 44,
        "gender": "Male",
        "faceBoundingBox": [
            {
                "x": 1601,
                "y": 395
            },
            {
                "x": 1653,
                "y": 395
            },
            {
                "x": 1653,
                "y": 447
            },
            {
                "x": 1601,
                "y": 447
            }
```

```
            ]
          }
        ],
        "objects": [
          {
            "rectangle": {
              "x": 25,
              "y": 43,
              "w": 172,
              "h": 140
            },
            "object": "person",
            "confidence": 0.931
          }
        ],
        "brands":[
           {
              "name":"Microsoft",
              "confidence": 0.903,
              "rectangle":{
                 "x":20,
                 "y":97,
                 "w":62,
                 "h":52
              }
           }
        ]
      }
    }
  ]
}
```


Error cases

In the following error cases, no elements are extracted.

Error Code	Description
`NotSupportedLanguage`	The language provided isn't supported.
`InvalidImageUrl`	Image URL is badly formatted or not accessible.
`InvalidImageFormat`	Input data isn't a valid image.
`InvalidImageSize`	Input image is too large.
`NotSupportedVisualFeature`	Specified feature type isn't valid.
`NotSupportedImage`	Unsupported image, for example, child pornography.
`InvalidDetails`	Unsupported domain-specific model.

If you get the error similar to `"One or more skills are invalid. Details: Error in skill #<num>: Outputs are not supported by skill: Landmarks"`, check the path. Both celebrities and landmarks are properties under `detail`.

JSON

```
"categories":[
      {
         "name":"building_",
         "score":0.97265625,
         "detail":{
            "landmarks":[
               {
                  "name":"Forbidden City",
                  "confidence":0.92013400793075562
               }
            ]
```

See also

- What is Image Analysis?
- Built-in skills
- How to define a skillset
- Extract text and information from images
- Create Indexer (REST)

Key Phrase Extraction cognitive skill

Article • 02/27/2023 • 2 minutes to read

The **Key Phrase Extraction** skill evaluates unstructured text, and for each record, returns a list of key phrases. This skill uses the Key Phrase machine learning models provided by Azure Cognitive Services for Language.

This capability is useful if you need to quickly identify the main talking points in the record. For example, given input text "The food was delicious and there were wonderful staff", the service returns "food" and "wonderful staff".

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.KeyPhraseExtractionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the key phrase extractor, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive.

Inputs	Description
`defaultLanguageCode`	(Optional) The language code to apply to documents that don't specify language explicitly. If the default language code is not specified, English (en) will be used as the default language code. See the full list of supported languages.
`maxKeyPhraseCount`	(Optional) The maximum number of key phrases to produce.

Inputs	Description
`modelVersion`	(Optional) Specifies the version of the model to use when calling the key phrase API. It will default to the latest available when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

Input	Description
`text`	The text to be analyzed.
`languageCode`	A string indicating the language of the records. If this parameter is not specified, the default language code will be used to analyze the records. See the full list of supported languages.

Skill outputs

Output	Description
`keyPhrases`	A list of key phrases extracted from the input text. The key phrases are returned in order of importance.

Sample definition

Consider a SQL record that has the following fields:

JSON

```
{
    "content": "Glaciers are huge rivers of ice that ooze their way over land, powered by gravity and their own sheer weight. They accumulate ice from snowfall and lose it through melting. As global temperatures have risen, many of the world’s glaciers have already started to shrink and retreat. Continued warming could see many iconic landscapes – from the Canadian Rockies to the Mount Everest region of the Himalayas – lose almost all their glaciers by the end of the century.",
    "language": "en"
}
```

Then your skill definition may look like this:

JSON

```
{
   "@odata.type": "#Microsoft.Skills.Text.KeyPhraseExtractionSkill",
   "inputs": [
     {
       "name": "text",
       "source": "/document/content"
     },
     {
       "name": "languageCode",
       "source": "/document/language"
     }
   ],
   "outputs": [
     {
       "name": "keyPhrases",
       "targetName": "myKeyPhrases"
     }
   ]
 }
```

Sample output

For the example above, the output of your skill will be written to a new node in the enriched tree called "document/myKeyPhrases" since that is the `targetName` that we specified. If you don't specify a `targetName`, then it would be "document/keyPhrases".

document/myKeyPhrases

JSON

```
[
  "world's glaciers",
  "huge rivers of ice",
  "Canadian Rockies",
  "iconic landscapes",
  "Mount Everest region",
  "Continued warming"
]
```

You may use "document/myKeyPhrases" as input into other skills, or as a source of an output field mapping.

Warnings

If you provide an unsupported language code, a warning is generated and key phrases are not extracted. If your text is empty, a warning will be produced. If your text is larger than 50,000 characters, only the first 50,000 characters will be analyzed and a warning will be issued.

See also

- Built-in skills
- How to define a skillset
- How to define output fields mappings

Language detection cognitive skill

Article • 10/04/2022 • 2 minutes to read

The **Language Detection** skill detects the language of input text and reports a single language code for every document submitted on the request. The language code is paired with a score indicating the strength of the analysis. This skill uses the machine learning models provided in Azure Cognitive Services for Language.

This capability is especially useful when you need to provide the language of the text as input to other skills (for example, the Sentiment Analysis skill or Text Split skill).

See supported languages for Language Detection. If you have content expressed in an unsupported language, the response is `(Unknown)`.

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.LanguageDetectionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the language detection skill, you may use the Text Split skill.

Skill parameters

Parameters are case-sensitive.

Inputs	Description

Inputs	Description
`defaultCountryHint`	(Optional) An ISO 3166-1 alpha-2 two letter country code can be provided to use as a hint to the language detection model if it cannot disambiguate the language. Specifically, the `defaultCountryHint` parameter is used with documents that don't specify the `countryHint` input explicitly.
`modelVersion`	(Optional) Specifies the version of the model to use when calling language detection. It will default to the latest available when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

Parameters are case-sensitive.

Inputs	Description
`text`	The text to be analyzed.
`countryHint`	An ISO 3166-1 alpha-2 two letter country code to use as a hint to the language detection model if it cannot disambiguate the language.

Skill outputs

Output Name	Description
`languageCode`	The ISO 6391 language code for the language identified. For example, "en".
`languageName`	The name of language. For example "English".
`score`	A value between 0 and 1. The likelihood that language is correctly identified. The score may be lower than 1 if the sentence has mixed languages.

Sample definition

JSON

```
{
   "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
   "inputs": [
     {
       "name": "text",
       "source": "/document/text"
     },
```

```
      {
        "name": "countryHint",
        "source": "/document/countryHint"
      }
    ],
    "outputs": [
      {
        "name": "languageCode",
        "targetName": "myLanguageCode"
      },
      {
        "name": "languageName",
        "targetName": "myLanguageName"
      },
      {
        "name": "score",
        "targetName": "myLanguageScore"
      }

    ]
  }
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "Glaciers are huge rivers of ice that ooze their way
over land, powered by gravity and their own sheer weight. "
           }
      },
      {
        "recordId": "2",
        "data":
           {
             "text": "Estamos muy felices de estar con ustedes."
           }
      },
      {
        "recordId": "3",
        "data":
           {
             "text": "impossible",
             "countryHint": "fr"
           }
      }
    ]
```

Sample output

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
            {
              "languageCode": "en",
              "languageName": "English",
              "score": 1,
            }
      },
      {
        "recordId": "2",
        "data":
            {
              "languageCode": "es",
              "languageName": "Spanish",
              "score": 1,
            }
      },
      {
        "recordId": "3",
        "data":
            {
              "languageCode": "fr",
              "languageName": "French",
              "score": 1,
            }
      }
    ]
}
```

See also

- Built-in skills
- How to define a skillset

OCR cognitive skill

Article • 10/12/2022 • 4 minutes to read

The **Optical character recognition (OCR)** skill recognizes printed and handwritten text in image files. This article is the reference documentation for the OCR skill. See Extract text from images for usage instructions.

An OCR skill uses the machine learning models provided by Computer Vision API v3.2 in Cognitive Services. The **OCR** skill maps to the following functionality:

- For the languages listed under Cognitive Services Computer Vision language support, the Read API is used.
- For Greek and Serbian Cyrillic, the legacy OCR API is used.

The **OCR** skill extracts text from image files. Supported file formats include:

- .JPEG
- .JPG
- .PNG
- .BMP
- .TIFF

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.
>
> In addition, image extraction is **billable by Azure Cognitive Search**.

Skill parameters

Parameters are case-sensitive.

Parameter name	Description
`detectOrientation`	Detects image orientation. Valid values are `true` or `false`. This parameter only applies if the legacy OCR API is used.

Parameter name	Description
`defaultLanguageCode`	Language code of the input text. Supported languages include all of the generally available languages of Cognitive Services Computer Vision. You can also specify `unk` (Unknown). If the language code is unspecified or null, the language is set to English. If the language is explicitly set to `unk`, all languages found are auto-detected and returned.
`lineEnding`	The value to use as a line separator. Possible values: "Space", "CarriageReturn", "LineFeed". The default is "Space".

In previous versions, there was a parameter called "textExtractionAlgorithm" to specify extraction of "printed" or "handwritten" text. This parameter is deprecated because the current Read API algorithm extracts both types of text at once. If your skill includes this parameter, you don't need to remove it, but it won't be used during skill execution.

Skill inputs

Input name	Description
`image`	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure Blob indexer when `imageAction` is set to a value other than `none`.

Skill outputs

Output name	Description
`text`	Plain text extracted from the image.
`layoutText`	Complex type that describes the extracted text and the location where the text was found.

If you call OCR on images embedded in PDFs or other application files, the OCR output will be located at the bottom of the page, after any text that was extracted and processed.

Sample definition

JSON

```
{
  "skills": [
    {
      "description": "Extracts text (plain and structured) from image.",
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
      "context": "/document/normalized_images/*",
      "defaultLanguageCode": null,
      "detectOrientation": true,
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [
        {
          "name": "text",
          "targetName": "myText"
        },
        {
          "name": "layoutText",
          "targetName": "myLayoutText"
        }
      ]
    }
  ]
}
```

Sample text and layoutText output

JSON

```
{
  "text": "Hello World. -John",
  "layoutText":
  {
    "language" : "en",
    "text" : "Hello World. -John",
    "lines" : [
      {
        "boundingBox":
        [ {"x":10, "y":10}, {"x":50, "y":10}, {"x":50, "y":30},{"x":10,
"y":30}],
        "text":"Hello World."
      },
      {
        "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150,
"y":30},{"x":110, "y":30}],
        "text":"-John"
      }
    ],
```

```
    "words": [
      {
        "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150,
"y":30},{"x":110, "y":30}],
        "text":"Hello"
      },
      {
        "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150,
"y":30},{"x":110, "y":30}],
        "text":"World."
      },
      {
        "boundingBox": [ {"x":110, "y":10}, {"x":150, "y":10}, {"x":150,
"y":30},{"x":110, "y":30}],
        "text":"-John"
      }
    ]
  }
}
```

Sample: Merging text extracted from embedded images with the content of the document

Document cracking, the first step in skillset execution, separates text and image content. A common use case for Text Merger is merging the textual representation of images (text from an OCR skill, or the caption of an image) into the content field of a document. This is for scenarios where the source document is a PDF or Word document that combines text with embedded images.

The following example skillset creates a *merged_text* field. This field contains the textual content of your document and the OCRed text from each of the images embedded in that document.

Request Body Syntax

JSON

```
{
  "description": "Extract text from images and merge with content text to
produce merged_text",
  "skills":
  [
    {
      "description": "Extract text (plain and structured) from image.",
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
```

```
      "context": "/document/normalized_images/*",
      "defaultLanguageCode": "en",
      "detectOrientation": true,
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [
        {
          "name": "text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
      "description": "Create merged_text, which includes all the textual
representation of each image inserted at the right location in the content
field.",
      "context": "/document",
      "insertPreTag": " ",
      "insertPostTag": " ",
      "inputs": [
        {
          "name":"text",
          "source": "/document/content"
        },
        {
          "name": "itemsToInsert",
          "source": "/document/normalized_images/*/text"
        },
        {
          "name":"offsets",
          "source": "/document/normalized_images/*/contentOffset"
        }
      ],
      "outputs": [
        {
          "name": "mergedText",
          "targetName" : "merged_text"
        }
      ]
    }
  ]
}
```

The above skillset example assumes that a normalized-images field exists. To generate this field, set the *imageAction* configuration in your indexer definition to *generateNormalizedImages* as shown below:

JSON

```
{
  //...rest of your indexer definition goes here ...
  "parameters": {
    "configuration": {
      "dataToExtract":"contentAndMetadata",
      "imageAction":"generateNormalizedImages"
    }
  }
}
```

See also

- What is optical character recognition
- Built-in skills
- TextMerger skill
- How to define a skillset
- Extract text and information from images
- Create Indexer (REST)

Additional resources

🕮 Documentation

Image Analysis cognitive skill - Azure Cognitive Search

Extract semantic text through image analysis using the Image Analysis cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Document Extraction cognitive skill - Azure Cognitive Search

Extracts content from a file within the enrichment pipeline.

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Extract text from images - Azure Cognitive Search

Use Optical Character Recognition (OCR) and image analysis to extract text, layout, captions, and tags from image files in Azure Cognitive Search pipelines.

Language detection cognitive skill - Azure Cognitive Search

Evaluates unstructured text, and for each record, returns a language identifier with a score indicating the strength of the analysis in an AI enrichment pipeline in Azure Cognitive Search.

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Entity Recognition cognitive skill (v2) - Azure Cognitive Search

Extract different types of entities from text in an enrichment pipeline in Azure Cognitive Search.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Show 5 more

Training

Module

OCR technologies in Power Automate for desktop - Training

This module covers the OCR actions in Power Automate for desktop.

Personally Identifiable Information (PII) Detection cognitive skill

Article • 02/27/2023 • 4 minutes to read

The **PII Detection** skill extracts personal information from an input text and gives you the option of masking it. This skill uses the detection models provided in Azure Cognitive Services for Language.

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.PIIDetectionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to chunk your data before sending it to the skill, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive and all are optional.

Parameter name	Description
`defaultLanguageCode`	(Optional) The language code to apply to documents that don't specify language explicitly. If the default language code is not specified, English (en) will be used as the default language code. See the full list of supported languages.
`minimumPrecision`	A value between 0.0 and 1.0. If the confidence score (in the `piiEntities` output) is lower than the set `minimumPrecision` value, the entity is not returned or masked. The default is 0.0.

Parameter name	Description
`maskingMode`	A parameter that provides various ways to mask the personal information detected in the input text. The following options are supported: • `"none"` (default): No masking occurs and the `maskedText` output will not be returned. • `"replace"`: Replaces the detected entities with the character given in the `maskingCharacter` parameter. The character will be repeated to the length of the detected entity so that the offsets will correctly correspond to both the input text and the output `maskedText`. When this skill was in public preview, the `maskingMode` option `redact` was also supported, which allowed removing the detected entities entirely without replacement. The `redact` option has since been deprecated and are longer be supported.
`maskingCharacter`	The character used to mask the text if the `maskingMode` parameter is set to `replace`. The following option is supported: `*` (default). This parameter can only be `null` if `maskingMode` is not set to `replace`. When this skill was in public preview, there was support for the `maskingCharacter` options, `X` and `#`. Both `X` and `#` options have since been deprecated and are longer be supported.
`domain`	(Optional) A string value, if specified, will set the domain to include only a subset of the entity categories. Possible values include: `"phi"` (detect confidential health information only), `"none"`.
`piiCategories`	(Optional) If you want to specify which entities will be detected and returned, use this optional parameter (defined as a list of strings) with the appropriate entity categories. This parameter can also let you detect entities that aren't enabled by default for your document language. See Supported Personally Identifiable Information entity categories for the full list.
`modelVersion`	(Optional) Specifies the version of the model to use when calling personally identifiable information detection. It will default to the most recent version when not specified. We recommend you do not specify this value unless it's necessary.

Skill inputs

Input name	Description
`languageCode`	A string indicating the language of the records. If this parameter is not specified, the default language code will be used to analyze the records. See the full list of supported languages.

Input name	Description
`text`	The text to analyze.

Skill outputs

Output name	Description
`piiEntities`	An array of complex types that contains the following fields: • `"text"` (The actual personally identifiable information as extracted) • `"type"` • `"subType"` • `"score"` (Higher value means it's more likely to be a real entity) • `"offset"` (into the input text) • `"length"` See Supported Personally Identifiable Information entity categories for the full list.
`maskedText`	If `maskingMode` is set to a value other than `none`, this output will be the string result of the masking performed on the input text as described by the selected `maskingMode`. If `maskingMode` is set to `none`, this output will not be present.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.PIIDetectionSkill",
  "defaultLanguageCode": "en",
  "minimumPrecision": 0.5,
  "maskingMode": "replace",
  "maskingCharacter": "*",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "piiEntities"
    },
    {
      "name": "maskedText"
    }
```

```
    ]
  }
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "Microsoft employee with ssn 859-98-0987 is using our
awesome API's."
           }
      }
    ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data" :
      {
        "piiEntities":[
           {
              "text":"859-98-0987",
              "type":"U.S. Social Security Number (SSN)",
              "subtype":"",
              "offset":28,
              "length":11,
              "score":0.65
           }
        ],
        "maskedText": "Microsoft employee with ssn *********** is using our
awesome API's."
      }
    }
  ]
}
```

The offsets returned for entities in the output of this skill are directly returned from the Language Service APIs, which means if you are using them to index into the original string, you should use the StringInfo class in .NET in order to extract the correct content. For more information, see Multilingual and emoji support in Language service features.

Errors and warnings

If the language code for the document is unsupported, a warning is returned and no entities are extracted. If your text is empty, a warning is returned. If your text is larger than 50,000 characters, only the first 50,000 characters will be analyzed and a warning will be issued.

If the skill returns a warning, the output `maskedText` may be empty, which can impact any downstream skills that expect the output. For this reason, be sure to investigate all warnings related to missing output when writing your skillset definition.

See also

- Built-in skills
- How to define a skillset

Additional resources

🕮 Documentation

Entity categories recognized by Personally Identifiable Information (detection) in Azure Cognitive Service for Language - Azure Cognitive Services

Learn about the entities the PII feature can recognize from unstructured text.

Azure Cognitive Services security - Azure Cognitive Services

Learn about the security considerations for Cognitive Services usage.

Data, privacy, and security for Azure Cognitive Service for language - Azure Cognitive Services

This document details issues for data and privacy for Azure Cognitive Service for language.

What is document and conversation summarization (preview)? - Azure Cognitive Services

Learn about summarizing text.

Azure Cognitive Services development options - Azure Cognitive Services

Learn how to use Azure Cognitive Services with different development and deployment options such

as client libraries, REST APIs, Logic Apps, Power Automate, Azure Functions, Azure App Service, Azure Databricks, and many more.

Plan to manage costs for Azure Cognitive Services - Azure Cognitive Services

Learn how to plan for and manage costs for Azure Cognitive Services by using cost analysis in the Azure portal.

What is the Personally Identifying Information (PII) detection feature in Azure Cognitive Service for Language? - Azure Cognitive Services

An overview of the PII detection feature in Azure Cognitive Services, which helps you extract entities and sensitive information (PII) in text.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Show 5 more

Sentiment cognitive skill (v3)

Article • 10/04/2022 • 3 minutes to read

The **Sentiment** skill (v3) evaluates unstructured text and for each record, provides sentiment labels (such as "negative", "neutral" and "positive") based on the highest confidence score found by the service at a sentence and document-level. This skill uses the machine learning models provided by version 3 of Language Service in Cognitive Services. It also exposes opinion mining capabilities, which provides more granular information about the opinions related to attributes of products or services in text.

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.

@odata.type

Microsoft.Skills.Text.V3.SentimentSkill

Data limits

The maximum size of a record should be 5000 characters as measured by String.Length. If you need to break up your data before sending it to the sentiment skill, use the Text Split skill.

Skill parameters

Parameters are case-sensitive.

Parameter Name	Description
`defaultLanguageCode`	(optional) The language code to apply to documents that don't specify language explicitly. See the full list of supported languages.
`modelVersion`	(optional) Specifies the version of the model to use when calling sentiment analysis. It will default to the most recent version when not specified. We recommend you do not specify this value unless it's necessary.

Parameter Name	Description
`includeOpinionMining`	If set to `true`, enables the opinion mining feature, which allows aspect-based sentiment analysis to be included in your output results. Defaults to `false`.

Skill inputs

Input Name	Description
`text`	The text to be analyzed.
`languageCode`	(optional) A string indicating the language of the records. If this parameter is not specified, the default value is "en". See the full list of supported languages.

Skill outputs

Output Name	Description
`sentiment`	A string value that represents the sentiment label of the entire analyzed text (either positive, neutral or negative).
`confidenceScores`	A complex type with three double values, one for the positive rating, one for the neutral rating, and one for the negative rating. Values range from 0 to 1.00, where 1.00 represents the highest possible confidence in a given label assignment.
`sentences`	A collection of complex types that breaks down the sentiment of the text sentence by sentence. This is also where opinion mining results are returned in the form of targets and assessments if `includeOpinionMining` is set to `true`.

Sample definition

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Text.V3.SentimentSkill",
    "context": "/document",
    "includeOpinionMining": true,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        },
```

```
        {
            "name": "languageCode",
            "source": "/document/languageCode"
        }
    ],
    "outputs": [
        {
            "name": "sentiment",
            "targetName": "sentiment"
        },
        {
            "name": "confidenceScores",
            "targetName": "confidenceScores"
        },
        {
            "name": "sentences",
            "targetName": "sentences"
        }
    ]
}
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "I had a terrible time at the hotel. The staff was
rude and the food was awful.",
                "languageCode": "en"
            }
        }
    ]
}
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "sentiment": "negative",
                "confidenceScores": {
```

```
                    "positive": 0.0,
                    "neutral": 0.0,
                    "negative": 1.0
                },
                "sentences": [
                    {
                        "text": "I had a terrible time at the hotel.",
                        "sentiment": "negative",
                        "confidenceScores": {
                            "positive": 0.0,
                            "neutral": 0.0,
                            "negative": 1.0
                        },
                        "offset": 0,
                        "length": 35,
                        "targets": [],
                        "assessments": [],
                    },
                    {
                        "text": "The staff was rude and the food was
awful.",
                        "sentiment": "negative",
                        "confidenceScores": {
                            "positive": 0.0,
                            "neutral": 0.0,
                            "negative": 1.0
                        },
                        "offset":36,
                        "length": 42,
                        "targets": [
                            {
                                "text": "staff",
                                "sentiment": "negative",
                                "confidenceScores": {
                                    "positive": 0.0,
                                    "neutral": 0.0,
                                    "negative": 1.0
                                },
                                "offset": 40,
                                "length": 5,
                                "relations": [
                                    {
                                        "relationType": "assessment",
                                        "ref":
"#/documents/0/sentences/1/assessments/0",
                                    }
                                ]
                            },
                            {
                                "text": "food",
                                "sentiment": "negative",
                                "confidenceScores": {
                                    "positive": 0.0,
                                    "neutral": 0.0,
                                    "negative": 1.0
```

```
                                    },
                                    "offset": 63,
                                    "length": 4,
                                    "relations": [
                                        {
                                            "relationType": "assessment",
                                            "ref":
"#/documents/0/sentences/1/assessments/1",
                                        }
                                    ]
                                }
                            ],
                            "assessments": [
                                {
                                    "text": "rude",
                                    "sentiment": "negative",
                                    "confidenceScores": {
                                        "positive": 0.0,
                                        "neutral": 0.0,
                                        "negative": 1.0
                                    },
                                    "offset": 50,
                                    "length": 4,
                                    "isNegated": false
                                },
                                {
                                    "text": "awful",
                                    "sentiment": "negative",
                                    "confidenceScores": {
                                        "positive": 0.0,
                                        "neutral": 0.0,
                                        "negative": 1.0
                                    },
                                    "offset": 72,
                                    "length": 5,
                                    "isNegated": false
                                }
                            ],
                        }
                    ]
                }
            }
        ]
}
```


Warning cases

If your text is empty, a warning is generated and no sentiment results are returned. If a language is not supported, a warning is generated and no sentiment results are returned.

See also

- Built-in skills
- How to define a skillset

Shaper cognitive skill

Article • 11/29/2021 • 4 minutes to read

The **Shaper** skill consolidates several inputs into a complex type that can be referenced later in the enrichment pipeline. The **Shaper** skill allows you to essentially create a structure, define the name of the members of that structure, and assign values to each member. Examples of consolidated fields useful in search scenarios include combining a first and last name into a single structure, city and state into a single structure, or name and birthdate into a single structure to establish unique identity.

Additionally, the **Shaper** skill illustrated in scenario 3 adds an optional *sourceContext* property to the input. The *source* and *sourceContext* properties are mutually exclusive. If the input is at the context of the skill, simply use *source*. If the input is at a *different* context than the skill context, use the *sourceContext*. The *sourceContext* requires you to define a nested input with the specific element being addressed as the source.

The output name is always "output". Internally, the pipeline can map a different name, such as "analyzedText" as shown in the examples below, but the **Shaper** skill itself returns "output" in the response. This might be important if you are debugging enriched documents and notice the naming discrepancy, or if you build a custom skill and are structuring the response yourself.

> **Note**
>
> This skill isn't bound to Cognitive Services. It is non-billable and has no Cognitive Services key requirement.

@odata.type

Microsoft.Skills.Util.ShaperSkill

Scenario 1: complex types

Consider a scenario where you want to create a structure called *analyzedText* that has two members: *text* and *sentiment*, respectively. In an index, a multi-part searchable field is called a *complex type* and it's often created when source data has a corresponding complex structure that maps to it.

However, another approach for creating complex types is through the **Shaper** skill. By including this skill in a skillset, the in-memory operations during skillset processing can output data shapes with nested structures, which can then be mapped to a complex type in your index.

The following example skill definition provides the member names as the input.

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
  "context": "/document/content/phrases/*",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content/phrases/*"
    },
    {
      "name": "sentiment",
      "source": "/document/content/phrases/*/sentiment"
    }
  ],
  "outputs": [
    {
      "name": "output",
      "targetName": "analyzedText"
    }
  ]
}
```

Sample index

A skillset is invoked by an indexer, and an indexer requires an index. A complex field representation in your index might look like the following example.

JSON

```
"name":"my-index",
"fields":[
   { "name":"myId", "type":"Edm.String", "key":true, "filterable":true  },
   { "name":"analyzedText", "type":"Edm.ComplexType",
      "fields":[
         {
            "name":"text",
            "type":"Edm.String",
            "facetable":false,
            "filterable":false,
            "searchable":true,
            "sortable":false  },
         {
```

```
            "name":"sentiment",
            "type":"Edm.Double",
            "facetable":true,
            "filterable":true,
            "searchable":true,
            "sortable":true }
      }
```

Skill input

An incoming JSON document providing usable input for this **Shaper** skill could be:

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "this movie is awesome",
                "sentiment": 0.9
            }
        }
    ]
}
```

Skill output

The **Shaper** skill generates a new element called *analyzedText* with the combined elements of *text* and *sentiment*. This output conforms to the index schema. It will be imported and indexed in an Azure Cognitive Search index.

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
            "analyzedText":
              {
                "text": "this movie is awesome" ,
                "sentiment": 0.9
              }
           }
      }
```

```
    ]
}
```

Scenario 2: input consolidation

In another example, imagine that at different stages of pipeline processing, you have extracted the title of a book, and chapter titles on different pages of the book. You could now create a single structure composed of these various inputs.

The **Shaper** skill definition for this scenario might look like the following example:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "context": "/document",
    "inputs": [
        {
            "name": "title",
            "source": "/document/content/title"
        },
        {
            "name": "chapterTitles",
            "source": "/document/content/pages/*/chapterTitles/*/title"
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "titlesAndChapters"
        }
    ]
}
```

Skill output

In this case, the **Shaper** flattens all chapter titles to create a single array.

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "titlesAndChapters": {
                    "title": "How to be happy",
                    "chapterTitles": [
```

```
                        "Start young",
                        "Laugh often",
                        "Eat, sleep and exercise"
                    ]
                }
            }
        }
    ]
}
```

Scenario 3: input consolidation from nested contexts

Imagine you have the title, chapters, and contents of a book and have run entity recognition and key phrases on the contents and now need to aggregate results from the different skills into a single shape with the chapter name, entities, and key phrases.

The **Shaper** skill definition for this scenario might look like the following example:

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "context": "/document",
    "inputs": [
        {
            "name": "title",
            "source": "/document/content/title"
        },
        {
            "name": "chapterTitles",
            "sourceContext": "/document/content/pages/*/chapterTitles/*",
            "inputs": [
              {
                  "name": "title",
                  "source":
"/document/content/pages/*/chapterTitles/*/title"
              },
              {
                  "name": "number",
                  "source":
"/document/content/pages/*/chapterTitles/*/number"
              }
            ]
        }

    ],
    "outputs": [
        {
            "name": "output",
```

```
            "targetName": "titlesAndChapters"
        }
    ]
}
```

Skill output

In this case, the **Shaper** creates a complex type. This structure exists in-memory. If you want to save it to a knowledge store, you should create a projection in your skillset that defines storage characteristics.

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "titlesAndChapters": {
                    "title": "How to be happy",
                    "chapterTitles": [
                      { "title": "Start young", "number": 1},
                      { "title": "Laugh often", "number": 2},
                      { "title": "Eat, sleep and exercise", "number: 3}
                    ]
                }
            }
        }
    ]
}
```

See also

- Built-in skills
- How to define a skillset
- How to use complex types
- Knowledge store
- Create a knowledge store in REST

Text Merge cognitive skill

Article • 04/21/2023

The **Text Merge** skill consolidates text from an array of strings into a single field.

> **Note**
>
> This skill isn't bound to Cognitive Services. It is non-billable and has no Cognitive Services key requirement.

@odata.type

Microsoft.Skills.Text.MergeSkill

Skill parameters

Parameters are case-sensitive.

Parameter name	Description
`insertPreTag`	String to be included before every insertion. The default value is `" "`. To omit the space, set the value to `""`.
`insertPostTag`	String to be included after every insertion. The default value is `" "`. To omit the space, set the value to `""`.

Skill inputs

Input name	Description
`itemsToInsert`	Array of strings to be merged.
`text`	(optional) Main text body to be inserted into. If `text` is not provided, elements of `itemsToInsert` will be concatencated.
`offsets`	(optional) Array of positions within `text` where `itemsToInsert` should be inserted. If provided, the number of elements of `text` must equal the number of elements of `textToInsert`. Otherwise all items will be appended at the end of `text`.

Skill outputs

Output name	Description
`mergedText`	The resulting merged text.
`mergedOffsets`	Array of positions within `mergedText` where elements of `itemsToInsert` were inserted.

Sample input

A JSON document providing usable input for this skill could be:

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data":
      {
        "text": "The brown fox jumps over the dog",
        "itemsToInsert": ["quick", "lazy"],
        "offsets": [3, 28]
      }
    }
  ]
}
```

Sample output

This example shows the output of the previous input, assuming that the *insertPreTag* is set to `" "`, and *insertPostTag* is set to `""`.

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data":
      {
        "mergedText": "The quick brown fox jumps over the lazy dog"
      }
    }
  ]
}
```

Extended sample skillset definition

A common scenario for using Text Merge is to merge the textual representation of images (text from an OCR skill, or the caption of an image) into the content field of a document.

The following example skillset uses the OCR skill to extract text from images embedded in the document. Next, it creates a *merged_text* field to contain both original and OCRed text from each image. You can learn more about the OCR skill here.

JSON

```
{
  "description": "Extract text from images and merge with content text to
produce merged_text",
  "skills":
  [
    {
      "description": "Extract text (plain and structured) from image.",
      "@odata.type": "#Microsoft.Skills.Vision.OcrSkill",
      "context": "/document/normalized_images/*",
      "defaultLanguageCode": "en",
      "detectOrientation": true,
      "inputs": [
        {
          "name": "image",
          "source": "/document/normalized_images/*"
        }
      ],
      "outputs": [
        {
          "name": "text"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.MergeSkill",
      "description": "Create merged_text, which includes all the textual
representation of each image inserted at the right location in the content
field.",
      "context": "/document",
      "insertPreTag": " ",
      "insertPostTag": " ",
      "inputs": [
        {
          "name":"text",
          "source": "/document/content"
        },
        {
```

```
          "name": "itemsToInsert",
          "source": "/document/normalized_images/*/text"
        },
        {
          "name":"offsets",
          "source": "/document/normalized_images/*/contentOffset"
        }
      ],
      "outputs": [
        {
          "name": "mergedText",
          "targetName" : "merged_text"
        }
      ]
    }
  ]
}
```

The example above assumes that a normalized-images field exists. To get normalized-images field, set the *imageAction* configuration in your indexer definition to *generateNormalizedImages* as shown below:

JSON

```
{
  //...rest of your indexer definition goes here ...
  "parameters":{
    "configuration":{
        "dataToExtract":"contentAndMetadata",
        "imageAction":"generateNormalizedImages"
    }
  }
}
```

See also

- Built-in skills
- How to define a skillset
- Create Indexer (REST)

Text split cognitive skill

Article • 02/27/2023 • 2 minutes to read

The **Text Split** skill breaks text into chunks of text. You can specify whether you want to break the text into sentences or into pages of a particular length. This skill is especially useful if there are maximum text length requirements in other skills downstream.

> ⓘ **Note**
>
> This skill isn't bound to Cognitive Services. It is non-billable and has no Cognitive Services key requirement.

@odata.type

Microsoft.Skills.Text.SplitSkill

Skill Parameters

Parameters are case-sensitive.

Parameter name	Description
`textSplitMode`	Either `pages` or `sentences`
`maximumPageLength`	Only applies if `textSplitMode` is set to `pages`. This refers to the maximum page length in characters as measured by `String.Length`. The minimum value is 300, the maximum is 100000, and the default value is 5000. The algorithm will do its best to break the text on sentence boundaries, so the size of each chunk may be slightly less than `maximumPageLength`.
`defaultLanguageCode`	(optional) One of the following language codes: `am, bs, cs, da, de, en, es, et, fr, he, hi, hr, hu, fi, id, is, it, ja, ko, lv, no, nl, pl, pt-PT, pt-BR, ru, sk, sl, sr, sv, tr, ur, zh-Hans`. Default is English (en). Few things to consider: • Providing a language code is useful to avoid cutting a word in half for non-whitespace languages such as Chinese, Japanese, and Korean. • If you do not know the language (i.e. you need to split the text for input into the LanguageDetectionSkill), the default of English (en) should be sufficient.

Skill Inputs

Parameter name	Description
`text`	The text to split into substring.
`languageCode`	(Optional) Language code for the document. If you do not know the language (i.e. you need to split the text for input into the LanguageDetectionSkill), it is safe to remove this input. If the language is not in the supported list for the `defaultLanguageCode` parameter above, a warning will be emitted and the text will not be split.

Skill Outputs

Parameter name	Description
`textItems`	An array of substrings that were extracted.

Sample definition

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
    "textSplitMode" : "pages",
    "maximumPageLength": 1000,
    "defaultLanguageCode": "en",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        },
        {
            "name": "languageCode",
            "source": "/document/language"
        }
    ],
    "outputs": [
        {
            "name": "textItems",
            "targetName": "mypages"
        }
    ]
}
```

Sample Input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "This is the loan application for Joe Romero, a
Microsoft employee who was born in Chile and who then moved to
Australia...",
                "languageCode": "en"
            }
        },
        {
            "recordId": "2",
            "data": {
                "text": "This is the second document, which will be broken
into several pages...",
                "languageCode": "en"
            }
        }
    ]
}
```

Sample Output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "textItems": [
                    "This is the loan…",
                    "On the second page we…"
                ]
            }
        },
        {
            "recordId": "2",
            "data": {
                "textItems": [
                    "This is the second document...",
                    "On the second page of the second doc…"
                ]
            }
        }
```

```
    ]
}
```

Error cases

If a language is not supported, a warning is generated.

See also

- Built-in skills
- How to define a skillset

Text Translation cognitive skill

Article • 02/27/2023 • 3 minutes to read

The **Text Translation** skill evaluates text and, for each record, returns the text translated to the specified target language. This skill uses the Translator Text API v3.0 available in Cognitive Services.

This capability is useful if you expect that your documents may not all be in one language, in which case you can normalize the text to a single language before indexing for search by translating it. It's also useful for localization use cases, where you may want to have copies of the same text available in multiple languages.

The Translator Text API v3.0 is a non-regional Cognitive Service, meaning that your data isn't guaranteed to stay in the same region as your Azure Cognitive Search or attached Cognitive Services resource.

> **Note**
>
> This skill is bound to Cognitive Services and requires **a billable resource** for transactions that exceed 20 documents per indexer per day. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.
>
> When using this skill, all documents in the source are processed and billed for translation, even if the source and target languages are the same. This behavior is useful for multi-language support within the same document, but it can result in unnecessary processing. To avoid unexpected billing charges from documents that don't need processing, move them out of the data source container prior to running the skill.

@odata.type

Microsoft.Skills.Text.TranslationSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the text translation skill, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive.

Inputs	Description
defaultToLanguageCode	(Required) The language code to translate documents into for documents that don't specify the "to" language explicitly. See the full list of supported languages.
defaultFromLanguageCode	(Optional) The language code to translate documents from for documents that don't specify the "from" language explicitly. If the defaultFromLanguageCode isn't specified, the automatic language detection provided by the Translator Text API will be used to determine the "from" language. See the full list of supported languages.
suggestedFrom	(Optional) The language code to translate documents from if `fromLanguageCode` or `defaultFromLanguageCode` are unspecified, and the automatic language detection is unsuccessful. If the suggestedFrom language isn't specified, English (en) will be used as the suggestedFrom language. See the full list of supported languages.

Skill inputs

Input name	Description
text	The text to be translated.
toLanguageCode	A string indicating the language the text should be translated to. If this input isn't specified, the defaultToLanguageCode will be used to translate the text. See the full list of supported languages.
fromLanguageCode	A string indicating the current language of the text. If this parameter isn't specified, the defaultFromLanguageCode (or automatic language detection if the defaultFromLanguageCode isn't provided) will be used to translate the text. See the full list of supported languages.

Skill outputs

Output name	Description

Output name	Description
translatedText	The string result of the text translation from the translatedFromLanguageCode to the translatedToLanguageCode.
translatedToLanguageCode	A string indicating the language code the text was translated to. Useful if you're translating to multiple languages and want to be able to keep track of which text is which language.
translatedFromLanguageCode	A string indicating the language code the text was translated from. Useful if you opted for the automatic language detection option as this output will give you the result of that detection.

Sample definition

JSON

```
{
   "@odata.type": "#Microsoft.Skills.Text.TranslationSkill",
   "defaultToLanguageCode": "fr",
   "suggestedFrom": "en",
   "context": "/document",
   "inputs": [
     {
       "name": "text",
       "source": "/document/text"
     }
   ],
   "outputs": [
     {
       "name": "translatedText",
       "targetName": "translatedText"
     },
     {
       "name": "translatedFromLanguageCode",
       "targetName": "translatedFromLanguageCode"
     },
     {
       "name": "translatedToLanguageCode",
       "targetName": "translatedToLanguageCode"
     }
   ]
 }
```

Sample input

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data":
        {
          "text": "We hold these truths to be self-evident, that all men are created equal."
        }
    },
    {
      "recordId": "2",
      "data":
        {
          "text": "Estamos muy felices de estar con ustedes."
        }
    }
  ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data":
        {
          "translatedText": "Nous tenons ces vérités pour évidentes, que tous les hommes sont créés égaux.",
          "translatedFromLanguageCode": "en",
          "translatedToLanguageCode": "fr"
        }
    },
    {
      "recordId": "2",
      "data":
        {
          "translatedText": "Nous sommes très heureux d'être avec vous.",
          "translatedFromLanguageCode": "es",
          "translatedToLanguageCode": "fr"
        }
    }
  ]
}
```

Errors and warnings

If you provide an unsupported language code for either the "to" or "from" language, an error is generated, and text isn't translated. If your text is empty, a warning will be produced. If your text is larger than 50,000 characters, only the first 50,000 characters will be translated, and a warning will be issued.

See also

- Built-in skills
- How to define a skillset

AML skill in an Azure Cognitive Search enrichment pipeline

Article • 12/02/2022 • 4 minutes to read

> **Important**
>
> This skill is in public preview under **supplemental terms of use**. The **preview REST API** supports this skill.

The **AML** skill allows you to extend AI enrichment with a custom Azure Machine Learning (AML) model. Once an AML model is trained and deployed, an **AML** skill integrates it into AI enrichment.

Like built-in skills, an **AML** skill has inputs and outputs. The inputs are sent to your deployed AML service as a JSON object, which outputs a JSON payload as a response along with a success status code. The response is expected to have the outputs specified by your **AML** skill. Any other response is considered an error and no enrichments are performed.

> **Note**
>
> The indexer will retry twice for certain standard HTTP status codes returned from the AML service. These HTTP status codes are:
>
> - `503 Service Unavailable`
> - `429 Too Many Requests`

Prerequisites

- An AML workspace
- An Azure Kubernetes Service AML compute target in this workspace with a deployed model
  - The compute target should have SSL enabled. Azure Cognitive Search only allows access to **https** endpoints
  - Self-signed certificates may not be used.

@odata.type

Microsoft.Skills.Custom.AmlSkill

Skill parameters

Parameters are case-sensitive. Which parameters you choose to use depends on what authentication your AML service requires, if any

Parameter name	Description
`uri`	(Required for no authentication or key authentication) The scoring URI of the AML service to which the *JSON* payload will be sent. Only the **https** URI scheme is allowed.
`key`	(Required for key authentication) The key for the AML service.
`resourceId`	(Required for token authentication). The Azure Resource Manager resource ID of the AML service. It should be in the format subscriptions/{guid}/resourceGroups/{resource-group-name}/Microsoft.MachineLearningServices/workspaces/{workspace-name}/services/{service_name}.
`region`	(Optional for token authentication). The region the AML service is deployed in.
`timeout`	(Optional) When specified, indicates the timeout for the http client making the API call. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). For example, `PT60S` for 60 seconds. If not set, a default value of 30 seconds is chosen. The timeout can be set to a maximum of 230 seconds and a minimum of 1 second.
`degreeOfParallelism`	(Optional) When specified, indicates the number of calls the indexer will make in parallel to the endpoint you have provided. You can decrease this value if your endpoint is failing under too high of a request load, or raise it if your endpoint is able to accept more requests and you would like an increase in the performance of the indexer. If not set, a default value of 5 is used. The degreeOfParallelism can be set to a maximum of 10 and a minimum of 1.

What skill parameters to use

Which AML skill parameters are required depends on what authentication your AML service uses, if any. AML services provide three authentication options:

- Key-Based Authentication. A static key is provided to authenticate scoring requests from AML skills
  - Use the *uri* and *key* parameters

- Token-Based Authentication. The AML service is deployed using token based authentication. The Azure Cognitive Search service's managed identity is granted the Reader Role in the AML service's workspace. The AML skill then uses the Azure Cognitive Search service's managed identity to authenticate against the AML service, with no static keys required.
  - Use the *resourceId* parameter.
  - If the Azure Cognitive Search service is in a different region from the AML workspace, use the *region* parameter to set the region the AML service was deployed in
- No Authentication. No authentication is required to use the AML service
  - Use the *uri* parameter

Skill inputs

There are no "predefined" inputs for this skill. You can choose one or more fields that would be already available at the time of this skill's execution as inputs and the *JSON* payload sent to the AML service will have different fields.

Skill outputs

There are no "predefined" outputs for this skill. Depending on the response your AML service will return, add output fields so that they can be picked up from the *JSON* response.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",
  "description": "A sample model that detects the language of sentence",
  "uri": "https://contoso.count-things.com/score",
  "context": "/document",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "detected_language_code"
    }
```

```
    ]
  }
```

Sample input JSON structure

This *JSON* structure represents the payload that will be sent to your AML service. The top-level fields of the structure will correspond to the "names" specified in the `inputs` section of the skill definition. The value of those fields will be from the `source` of those fields (which could be from a field in the document, or potentially from another skill)

JSON

```
{
  "text": "Este es un contrato en Inglés"
}
```

Sample output JSON structure

The output corresponds to the response returned from your AML service. The AML service should only return a *JSON* payload (verified by looking at the `Content-Type` response header) and should be an object where the fields are enrichments matching the "names" in the `output` and whose value is considered the enrichment.

JSON

```
{
    "detected_language_code": "es"
}
```

Inline shaping sample definition

JSON

```
  {
    "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",
    "description": "A sample model that detects the language of sentence",
    "uri": "https://contoso.count-things.com/score",
    "context": "/document",
    "inputs": [
      {
        "name": "shapedText",
        "sourceContext": "/document",
        "inputs": [
```

```
            {
              "name": "content",
              "source": "/document/content"
            }
          ]
        }
      ],
      "outputs": [
        {
          "name": "detected_language_code"
        }
      ]
    }
```

Inline shaping input JSON structure

JSON

```
{
  "shapedText": { "content": "Este es un contrato en Inglés" }
}
```

Inline shaping sample output JSON structure

JSON

```
{
    "detected_language_code": "es"
}
```

Error cases

In addition to your AML being unavailable or sending out non-successful status codes, the following are considered erroneous cases:

- If the AML service returns a success status code but the response indicates that it is not `application/json`, then the response is considered invalid and no enrichments will be performed.
- If the AML service returns invalid json

For cases when the AML service is unavailable or returns an HTTP error, a friendly error with any available details about the HTTP error will be added to the indexer execution history.

See also

- How to define a skillset
- AML Service troubleshooting

Custom Web API skill in an Azure Cognitive Search enrichment pipeline

Article • 08/23/2022 • 6 minutes to read

The **Custom Web API** skill allows you to extend AI enrichment by calling out to a Web API endpoint providing custom operations. Similar to built-in skills, a **Custom Web API** skill has inputs and outputs. Depending on the inputs, your Web API receives a JSON payload when the indexer runs, and outputs a JSON payload as a response, along with a success status code. The response is expected to have the outputs specified by your custom skill. Any other response is considered an error and no enrichments are performed.

The structure of the JSON payload is described further down in this document.

> **Note**
>
> The indexer will retry twice for certain standard HTTP status codes returned from the Web API. These HTTP status codes are:
>
> - `502 Bad Gateway`
> - `503 Service Unavailable`
> - `429 Too Many Requests`

@odata.type

Microsoft.Skills.Custom.WebApiSkill

Skill parameters

Parameters are case-sensitive.

Parameter name	Description
`uri`	The URI of the Web API to which the JSON payload will be sent. Only the **https** URI scheme is allowed.

Parameter name	Description
`authResourceId`	(Optional) A string that if set, indicates that this skill should use a managed identity on the connection to the function or app hosting the code. The value of this property is the application (client) ID of the function or app's registration in Azure Active Directory. This value will be used to scope the authentication token retrieved by the indexer, and will be sent along with the custom Web skill API request to the function or app. Setting this property requires that your search service is configured for managed identity and your Azure function app is configured for an Azure AD login.
`httpMethod`	The method to use while sending the payload. Allowed methods are `PUT` or `POST`
`httpHeaders`	A collection of key-value pairs where the keys represent header names and values represent header values that will be sent to your Web API along with the payload. The following headers are prohibited from being in this collection: `Accept`, `Accept-Charset`, `Accept-Encoding`, `Content-Length`, `Content-Type`, `Cookie`, `Host`, `TE`, `Upgrade`, `Via`.
`timeout`	(Optional) When specified, indicates the timeout for the http client making the API call. It must be formatted as an XSD "dayTimeDuration" value (a restricted subset of an ISO 8601 duration value). For example, `PT60S` for 60 seconds. If not set, a default value of 30 seconds is chosen. The timeout can be set to a maximum of 230 seconds and a minimum of 1 second.
`batchSize`	(Optional) Indicates how many "data records" (see JSON payload structure below) will be sent per API call. If not set, a default of 1000 is chosen. We recommend that you make use of this parameter to achieve a suitable tradeoff between indexing throughput and load on your API.
`degreeOfParallelism`	(Optional) When specified, indicates the number of calls the indexer will make in parallel to the endpoint you have provided. You can decrease this value if your endpoint is failing under pressure, or raise it if your endpoint can handle the load. If not set, a default value of 5 is used. The `degreeOfParallelism` can be set to a maximum of 10 and a minimum of 1.

Skill inputs

There are no predefined inputs for this skill. The inputs are any existing field, or any node in the enrichment tree that you want to pass to your custom skill.

Skill outputs

There are no predefined outputs for this skill. Be sure to define an output field mapping in the indexer if the skill's output should be sent to a field in the search index.

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
  "description": "A custom skill that can identify positions of different
phrases in the source text",
  "uri": "https://contoso.count-things.com",
  "batchSize": 4,
  "context": "/document",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    },
    {
      "name": "language",
      "source": "/document/languageCode"
    },
    {
      "name": "phraseList",
      "source": "/document/keyphrases"
    }
  ],
  "outputs": [
    {
      "name": "hitPositions"
    }
  ]
}
```

Sample input JSON structure

This JSON structure represents the payload that will be sent to your Web API. It will always follow these constraints:

- The top-level entity is called `values` and will be an array of objects. The number of such objects will be at most the `batchSize`.
- Each object in the `values` array will have:
  - A `recordId` property that is a **unique** string, used to identify that record.

- A `data` property that is a JSON object. The fields of the `data` property will correspond to the "names" specified in the `inputs` section of the skill definition. The value of those fields will be from the `source` of those fields (which could be from a field in the document, or potentially from another skill).

JSON

```
{
    "values": [
      {
        "recordId": "0",
        "data":
           {
             "text": "Este es un contrato en Inglés",
             "language": "es",
             "phraseList": ["Este", "Inglés"]
           }
      },
      {
        "recordId": "1",
        "data":
           {
             "text": "Hello world",
             "language": "en",
             "phraseList": ["Hi"]
           }
      },
      {
        "recordId": "2",
        "data":
           {
             "text": "Hello world, Hi world",
             "language": "en",
             "phraseList": ["world"]
           }
      },
      {
        "recordId": "3",
        "data":
           {
             "text": "Test",
             "language": "es",
             "phraseList": []
           }
      }
    ]
}
```

Sample output JSON structure

The "output" corresponds to the response returned from your Web API. The Web API should only return a JSON payload (verified by looking at the `Content-Type` response header) and should satisfy the following constraints:

- There should be a top-level entity called `values` which should be an array of objects.
- The number of objects in the array should be the same as the number of objects sent to the Web API.
- Each object should have:
  - A `recordId` property.
  - A `data` property, which is an object where the fields are enrichments matching the "names" in the `output` and whose value is considered the enrichment.
  - An `errors` property, an array listing any errors encountered that will be added to the indexer execution history. This property is required, but can have a `null` value.
  - A `warnings` property, an array listing any warnings encountered that will be added to the indexer execution history. This property is required, but can have a `null` value.
- The ordering of objects in the `values` in either the request or response isn't important. However, the `recordId` is used for correlation so any record in the response containing a `recordId` which was not part of the original request to the Web API will be discarded.

JSON

```
{
    "values": [
        {
            "recordId": "3",
            "data": {
            },
            "errors": [
              {
                "message" : "'phraseList' should not be null or empty"
              }
            ],
            "warnings": null
        },
        {
            "recordId": "2",
```

```
            "data": {
                "hitPositions": [6, 16]
            },
            "errors": null,
            "warnings": null
        },
        {
            "recordId": "0",
            "data": {
                "hitPositions": [0, 23]
            },
            "errors": null,
            "warnings": null
        },
        {
            "recordId": "1",
            "data": {
                "hitPositions": []
            },
            "errors": null,
            "warnings": {
                "message": "No occurrences of 'Hi' were found in the input
text"
            }
        },
    ]
}
```

Error cases

In addition to your Web API being unavailable, or sending out non-successful status codes the following are considered erroneous cases:

- If the Web API returns a success status code but the response indicates that it is not `application/json` then the response is considered invalid and no enrichments will be performed.
- If there are invalid records (for example, `recordId` is missing or duplicated) in the response `values` array, no enrichment will be performed for the invalid records.

For cases when the Web API is unavailable or returns an HTTP error, a friendly error with any available details about the HTTP error will be added to the indexer execution history.

See also

- How to define a skillset
- Add custom skill to an AI enrichment pipeline
- Example: Creating a custom skill for AI enrichment

Additional resources

🕮 Documentation

Shaper cognitive skill - Azure Cognitive Search

Extract metadata and structured information from unstructured data and shape it as a complex type in an AI enrichment pipeline in Azure Cognitive Search.

Create a knowledge store using REST - Azure Cognitive Search

Use the REST API and Postman to create an Azure Cognitive Search knowledge store for persisting AI enrichments from skillset.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Text Merge cognitive skill - Azure Cognitive Search

Merge text from a collection of fields into one consolidated field. Use this cognitive skill in an AI enrichment pipeline in Azure Cognitive Search.

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Show 5 more

Deprecated Cognitive Skills in Azure Cognitive Search

Article • 08/18/2022 • 8 minutes to read

This document describes cognitive skills that are considered deprecated (retired). Use the following guide for the contents:

- Skill Name: The name of the skill that will be deprecated; it maps to the @odata.type attribute.
- Last available api version: The last version of the Azure Cognitive Search public API through which skillsets containing the corresponding deprecated skill can be created/updated. Indexers with attached skillsets with these skills will continue to run even in future API versions until the "End of support" date, at which point they will start failing.
- End of support: The day after which the corresponding skill is considered unsupported and will stop working. Previously created skillsets should still continue to function, but users are recommended to migrate away from a deprecated skill.
- Recommendations: Migration path forward to use a supported skill. Users are advised to follow the recommendations to continue to receive support.

If you're using the Microsoft.Skills.Text.EntityRecognitionSkill (Entity Recognition cognitive skill (v2)), this article will help you upgrade your skillset to use the Microsoft.Skills.Text.V3.EntityRecognitionSkill which is generally available and introduces new features.

If you're using the Microsoft.Skills.Text.SentimentSkill (Sentiment cognitive skill (v2)), this article will help you upgrade your skillset to use the Microsoft.Skills.Text.V3.SentimentSkill which is generally available and introduces new features.

If you're using the Microsoft.Skills.Text.NamedEntityRecognitionSkill (Named Entity Recognition cognitive skill (v2)), this article will help you upgrade your skillset to use the Microsoft.Skills.Text.V3.EntityRecognitionSkill which is generally available and introduces new features.

Microsoft.Skills.Text.EntityRecognitionSkill

Last available api version

2021-04-30-Preview

End of support

August 31, 2024

Recommendations

Use Microsoft.Skills.Text.V3.EntityRecognitionSkill instead. It provides most of the functionality of the EntityRecognitionSkill at a higher quality. It also has richer information in its complex output fields.

To migrate to the Microsoft.Skills.Text.V3.EntityRecognitionSkill, you will have to perform one or more of the following changes to your skill definition. You can update the skill definition using the Update Skillset API.

1. *(Required)* Change the `@odata.type` from `"#Microsoft.Skills.Text.EntityRecognitionSkill"` to `"#Microsoft.Skills.Text.V3.EntityRecognitionSkill"`.

2. *(Optional)* The `includeTypelessEntities` parameter is no longer supported as the new skill will only ever return entities with known types, so if your previous skill definition referenced it, it should now be removed.

3. *(Optional)* If you are making use of the `namedEntities` output, there are a few minor changes to the property names.
   a. `value` is renamed to `text`
   b. `confidence` is renamed to `confidenceScore`

   If you need to generate the exact same property names, you will need to add a ShaperSkill to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

   JSON

   ```
   {
       "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
       "name": "NamedEntitiesShaper",
       "description": "NamedEntitiesShaper",
       "context": "/document/namedEntitiesV3",
       "inputs": [
           {
               "name": "old_format",
               "sourceContext": "/document/namedEntitiesV3/*",
               "inputs": [
   ```

```
                {
                    "name": "value",
                    "source": "/document/namedEntitiesV3/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntitiesV3/*/offset"
                },
                {
                    "name": "category",
                    "source": "/document/namedEntitiesV3/*/category"
                },
                {
                    "name": "confidence",
                    "source":
"/document/namedEntitiesV3/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "namedEntities"
        }
    ]
}
```

4. *(Optional)* If you are making use of the `entities` output to link entities to well-known entities, this feature is now a new skill, the Microsoft.Skills.Text.V3.EntityLinkingSkill. Add the entity linking skill to your skillset to generate the linked entities. There are also a few minor changes to the property names of the `entities` output between the `EntityRecognitionSkill` and the new `EntityLinkingSkill`.
   a. `wikipediaId` is renamed to `id`
   b. `wikipediaLanguage` is renamed to `language`
   c. `wikipediaUrl` is renamed to `url`
   d. The `type` and `subtype` properties are no longer returned.

   If you need to generate the exact same property names, you will need to add a ShaperSkill to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "LinkedEntitiesShaper",
```

```
    "description": "LinkedEntitiesShaper",
    "context": "/document/linkedEntitiesV3",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/linkedEntitiesV3/*",
            "inputs": [
                {
                    "name": "name",
                    "source": "/document/linkedEntitiesV3/*/name"
                },
                {
                    "name": "wikipediaId",
                    "source": "/document/linkedEntitiesV3/*/id"
                },
                {
                    "name": "wikipediaLanguage",
                    "source": "/document/linkedEntitiesV3/*/language"
                },
                {
                    "name": "wikipediaUrl",
                    "source": "/document/linkedEntitiesV3/*/url"
                },
                {
                    "name": "bingId",
                    "source": "/document/linkedEntitiesV3/*/bingId"
                },
                {
                    "name": "matches",
                    "source": "/document/linkedEntitiesV3/*/matches"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}
```

5. *(Optional)* If you do not explicitly specify the `categories`, the `EntityRecognitionSkill V3` can return different type of categories besides those that were supported by the `EntityRecognitionSkill`. If this behavior is undesirable, make sure to explicitly set the `categories` parameter to `["Person", "Location", "Organization", "Quantity", "Datetime", "URL", "Email"]`.

Sample Migration Definitions

- Simple migration

(Before) EntityRecognition skill definition

JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.EntityRecognitionSkill",
    "categories": [ "Person" ],
    "defaultLanguageCode": "en",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        }
    ]
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "categories": [ "Person" ],
    "defaultLanguageCode": "en",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        }
    ]
}
```

- Complicated migration

(Before) EntityRecognition skill definition

JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.EntityRecognitionSkill",
    "categories": [ "Person", "Location", "Organization" ],
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "includeTypelessEntities": true,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        },
        {
            "name": "namedEntities",
            "targetName": "namedEntities"
        },
        {
            "name": "entities",
            "targetName": "entities"
        }
    ]
}
```

(After) EntityRecognition skill V3 definition

JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "categories": [ "Person", "Location", "Organization" ],
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        },
        {
            "name": "namedEntities",
```

```
            "targetName": "namedEntitiesV3"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "NamedEntitiesShaper",
    "description": "NamedEntitiesShaper",
    "context": "/document/namedEntitiesV3",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/namedEntitiesV3/*",
            "inputs": [
                {
                    "name": "value",
                    "source": "/document/namedEntitiesV3/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntitiesV3/*/offset"
                },
                {
                    "name": "category",
                    "source":
"/document/namedEntitiesV3/*/category"
                },
                {
                    "name": "confidence",
                    "source":
"/document/namedEntitiesV3/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "namedEntities"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Text.V3.EntityLinkingSkill",
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "entities",
```

```
            "targetName": "linkedEntities"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "LinkedEntitiesShaper",
    "description": "LinkedEntitiesShaper",
    "context": "/document/linkedEntitiesV3",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/linkedEntitiesV3/*",
            "inputs": [
                {
                    "name": "name",
                    "source": "/document/linkedEntitiesV3/*/name"
                },
                {
                    "name": "wikipediaId",
                    "source": "/document/linkedEntitiesV3/*/id"
                },
                {
                    "name": "wikipediaLanguage",
                    "source":
"/document/linkedEntitiesV3/*/language"
                },
                {
                    "name": "wikipediaUrl",
                    "source": "/document/linkedEntitiesV3/*/url"
                },
                {
                    "name": "bingId",
                    "source":
"/document/linkedEntitiesV3/*/bingId"
                },
                {
                    "name": "matches",
                    "source":
"/document/linkedEntitiesV3/*/matches"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}
```

Microsoft.Skills.Text.SentimentSkill

Last available api version

2021-04-30-Preview

End of support

August 31, 2024

Recommendations

Use Microsoft.Skills.Text.V3.SentimentSkill instead. It provides an improved model and includes the option to add opinion mining or aspect-based sentiment. As the skill is significantly more complex, the outputs are also very different.

To migrate to the Microsoft.Skills.Text.V3.SentimentSkill, you will have to perform one or more of the following changes to your skill definition. You can update the skill definition using the Update Skillset API.

> **Note**
>
> The skill outputs for the Sentiment Skill V3 are not compatible with the index definition based on the SentimentSkill. You will have to make changes to the index definition, skillset (later skill inputs and/or knowledge store projections) and indexer output field mappings to replace the sentiment skill with the new version.

1. *(Required)* Change the `@odata.type` from `"#Microsoft.Skills.Text.SentimentSkill"` to `"#Microsoft.Skills.Text.V3.SentimentSkill"`.

2. *(Required)* The Sentiment Skill V3 provides a `positive`, `neutral`, and `negative` score for the overall text and the same scores for each sentence in the overall text, whereas the previous SentimentSkill only provided a single double that ranged from 0.0 (negative) to 1.0 (positive) for the overall text. You will need to update your index definition to accept the three double values in place of a single score, and make sure all of your downstream skill inputs, knowledge store projections, and output field mappings are consistent with the naming changes.

It is recommended to replace the old SentimentSkill with the SentimentSkill V3 entirely, update your downstream skill inputs, knowledge store projections, indexer output field mappings, and index definition to match the new output format, and reset your indexer so that all of your documents have consistent sentiment results going forward.

> **Note**
>
> If you need any additional help updating your enrichment pipeline to use the latest version of the sentiment skill or if resetting your indexer is not an option for you, please open a **new support request** where we can work with you directly.

Microsoft.Skills.Text.NamedEntityRecognitionSkill

Last available api version

2017-11-11-Preview

End of support

August 31, 2024

Recommendations

Use Microsoft.Skills.Text.V3.EntityRecognitionSkill instead. It provides most of the functionality of the NamedEntityRecognitionSkill at a higher quality. It also has richer information in its complex output fields.

To migrate to the Microsoft.Skills.Text.V3.EntityRecognitionSkill, you will have to perform one or more of the following changes to your skill definition. You can update the skill definition using the Update Skillset API.

1. *(Required)* Change the `@odata.type` from `"#Microsoft.Skills.Text.NamedEntityRecognitionSkill"` to `"#Microsoft.Skills.Text.V3.EntityRecognitionSkill"`.

2. *(Optional)* If you are making use of the `entities` output, use the `namedEntities` complex collection output from the `EntityRecognitionSkill V3` instead. There are a few minor changes to the property names of the new `namedEntities` complex output:
   a. `value` is renamed to `text`

b. `confidence` is renamed to `confidenceScore`

If you need to generate the exact same property names, you will need to add a ShaperSkill to reshape the output with the required names. For example, this ShaperSkill renames the properties to their old values.

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "NamedEntitiesShaper",
    "description": "NamedEntitiesShaper",
    "context": "/document/namedEntities",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/namedEntities/*",
            "inputs": [
                {
                    "name": "value",
                    "source": "/document/namedEntities/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntities/*/offset"
                },
                {
                    "name": "category",
                    "source": "/document/namedEntities/*/category"
                },
                {
                    "name": "confidence",
                    "source":
"/document/namedEntities/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}
```

3. *(Optional)* If you do not explicitly specify the `categories`, the `EntityRecognitionSkill V3` can return different type of categories besides those that were supported by the `NamedEntityRecognitionSkill`. If this behavior is

undesirable, make sure to explicitly set the `categories` parameter to `["Person", "Location", "Organization"]`.

Sample Migration Definitions

- Simple migration

  (Before) NamedEntityRecognition skill definition

  JSON

  ```
  {
      "@odata.type":
  "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",
      "categories": [ "Person" ],
      "defaultLanguageCode": "en",
      "inputs": [
          {
              "name": "text",
              "source": "/document/content"
          }
      ],
      "outputs": [
          {
              "name": "persons",
              "targetName": "people"
          }
      ]
  }
  ```

  (After) EntityRecognition skill V3 definition

  JSON

  ```
  {
      "@odata.type":
  "#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
      "categories": [ "Person" ],
      "defaultLanguageCode": "en",
      "inputs": [
          {
              "name": "text",
              "source": "/document/content"
          }
      ],
      "outputs": [
          {
              "name": "persons",
              "targetName": "people"
          }
  ```

```
    ]
}
```

- Slightly complicated migration

  (Before) NamedEntityRecognition skill definition

  JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.NamedEntityRecognitionSkill",
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
        },
        {
            "name": "entities"
        }
    ]
}
```

  (After) EntityRecognition skill V3 definition

  JSON

```
{
    "@odata.type":
"#Microsoft.Skills.Text.V3.EntityRecognitionSkill",
    "categories": [ "Person", "Location", "Organization" ],
    "defaultLanguageCode": "en",
    "minimumPrecision": 0.1,
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        }
    ],
    "outputs": [
        {
            "name": "persons",
            "targetName": "people"
```

```
        },
        {
            "name": "namedEntities"
        }
    ]
},
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "NamedEntitiesShaper",
    "description": "NamedEntitiesShaper",
    "context": "/document/namedEntities",
    "inputs": [
        {
            "name": "old_format",
            "sourceContext": "/document/namedEntities/*",
            "inputs": [
                {
                    "name": "value",
                    "source": "/document/namedEntities/*/text"
                },
                {
                    "name": "offset",
                    "source": "/document/namedEntities/*/offset"
                },
                {
                    "name": "category",
                    "source": "/document/namedEntities/*/category"
                },
                {
                    "name": "confidence",
                    "source":
"/document/namedEntities/*/confidenceScore"
                }
            ]
        }
    ],
    "outputs": [
        {
            "name": "output",
            "targetName": "entities"
        }
    ]
}
```

See also

- Built-in skills
- How to define a skillset
- Entity Recognition Skill (V3)
- Sentiment Skill (V3)

Named Entity Recognition cognitive skill (v2)

Article • 02/27/2023 • 2 minutes to read

The **Named Entity Recognition** skill (v2) extracts named entities from text. Available entities include the types `person`, `location` and `organization`.

> **Important**
>
> Named entity recognition skill (v2) (**Microsoft.Skills.Text.NamedEntityRecognitionSkill**) is now discontinued replaced by **Microsoft.Skills.Text.V3.EntityRecognitionSkill**. Follow the recommendations in **Deprecated cognitive search skills** to migrate to a supported skill.

> **Note**
>
> As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to **attach a billable Cognitive Services resource**. Charges accrue when calling APIs in Cognitive Services, and for image extraction as part of the document-cracking stage in Azure Cognitive Search. There are no charges for text extraction from documents. Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**.
>
> Image extraction is an extra charge metered by Azure Cognitive Search, as described on the **pricing page**. Text extraction is free.

@odata.type

Microsoft.Skills.Text.NamedEntityRecognitionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the key phrase extractor, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive.

Parameter name	Description
categories	Array of categories that should be extracted. Possible category types: `"Person"`, `"Location"`, `"Organization"`. If no category is provided, all types are returned.
defaultLanguageCode	Language code of the input text. The following languages are supported: `de, en, es, fr, it`
minimumPrecision	A number between 0 and 1. If the precision is lower than this value, the entity is not returned. The default is 0.

Skill inputs

Input name	Description
languageCode	Optional. Default is `"en"`.
text	The text to analyze.

Skill outputs

Output name	Description
persons	An array of strings where each string represents the name of a person.
locations	An array of strings where each string represents a location.
organizations	An array of strings where each string represents an organization.
entities	An array of complex types. Each complex type includes the following fields: • category (`"person"`, `"organization"`, or `"location"`) • value (the actual entity name) • offset (The location where it was found in the text) • confidence (A value between 0 and 1 that represents that confidence that the value is an actual entity)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",
  "categories": [ "Person", "Location", "Organization"],
  "defaultLanguageCode": "en",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    }
  ]
}
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "This is the loan application for Joe Romero, a
Microsoft employee who was born in Chile and who then moved to Australia…
Ana Smith is provided as a reference.",
             "languageCode": "en"
           }
      }
    ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data" :
```

```
      {
        "persons": [ "Joe Romero", "Ana Smith"],
        "locations": ["Chile", "Australia"],
        "organizations":["Microsoft"],
        "entities":
        [
          {
            "category":"person",
            "value": "Joe Romero",
            "offset": 33,
            "confidence": 0.87
          },
          {
            "category":"person",
            "value": "Ana Smith",
            "offset": 124,
            "confidence": 0.87
          },
          {
            "category":"location",
            "value": "Chile",
            "offset": 88,
            "confidence": 0.99
          },
          {
            "category":"location",
            "value": "Australia",
            "offset": 112,
            "confidence": 0.99
          },
          {
            "category":"organization",
            "value": "Microsoft",
            "offset": 54,
            "confidence": 0.99
          }
        ]
      }
    }
  ]
}
```

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- Built-in skills

- How to define a skillset
- Entity Recognition Skill (V3)

Additional resources

Documentation

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Reference inputs and outputs in skillsets - Azure Cognitive Search

Explains the annotation syntax and how to reference inputs and outputs of a skillset in an AI enrichment pipeline in Azure Cognitive Search.

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Show 5 more

Training

Learning paths and modules

Improve communication skills with the 21CLD skilled communication dimension - Training

This module introduces the concept of skilled communication to educators and supports them in

designing learning activities to help young people develop this important skill.

Entity Recognition cognitive skill (v2)

Article • 02/27/2023 • 4 minutes to read

The **Entity Recognition** skill (v2) extracts entities of different types from text. This skill uses the machine learning models provided by Text Analytics in Cognitive Services.

> **Important**
>
> The Entity Recognition skill (v2) (**Microsoft.Skills.Text.EntityRecognitionSkill**) is now discontinued replaced by **Microsoft.Skills.Text.V3.EntityRecognitionSkill**. Follow the recommendations in **Deprecated cognitive search skills** to migrate to a supported skill.

> **Note**
>
> As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to **attach a billable Cognitive Services resource**. Charges accrue when calling APIs in Cognitive Services, and for image extraction as part of the document-cracking stage in Azure Cognitive Search. There are no charges for text extraction from documents.
>
> Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**. Image extraction pricing is described on the **Azure Cognitive Search pricing page**.

@odata.type

Microsoft.Skills.Text.EntityRecognitionSkill

Data limits

The maximum size of a record should be 50,000 characters as measured by String.Length. If you need to break up your data before sending it to the key phrase extractor, consider using the Text Split skill. If you do use a text split skill, set the page length to 5000 for the best performance.

Skill parameters

Parameters are case-sensitive and are all optional.

Parameter name	Description
`categories`	Array of categories that should be extracted. Possible category types: `"Person"`, `"Location"`, `"Organization"`, `"Quantity"`, `"Datetime"`, `"URL"`, `"Email"`. If no category is provided, all types are returned.
`defaultLanguageCode`	Language code of the input text. The following languages are supported: `ar, cs, da, de, en, es, fi, fr, hu, it, ja, ko, nl, no, pl, pt-BR, pt-PT, ru, sv, tr, zh-hans`. Not all entity categories are supported for all languages; see note below.
`minimumPrecision`	A value between 0 and 1. If the confidence score (in the `namedEntities` output) is lower than this value, the entity is not returned. The default is 0.
`includeTypelessEntities`	Set to `true` if you want to recognize well-known entities that don't fit the current categories. Recognized entities are returned in the `entities` complex output field. For example, "Windows 10" is a well-known entity (a product), but since "Products" is not a supported category, this entity would be included in the entities output field. Default is `false`

Skill inputs

Input name	Description
`languageCode`	Optional. Default is `"en"`.
`text`	The text to analyze.

Skill outputs

> **Note**
>
> Not all entity categories are supported for all languages. The `"Person"`, `"Location"`, and `"Organization"` entity category types are supported for the full list of languages above. Only *de*, *en*, *es*, *fr*, and *zh-hans* support extraction of `"Quantity"`, `"Datetime"`, `"URL"`, and `"Email"` types. For more information, see **Language and region support for the Text Analytics API**.

Output name	Description

Output name	Description
`persons`	An array of strings where each string represents the name of a person.
`locations`	An array of strings where each string represents a location.
`organizations`	An array of strings where each string represents an organization.
`quantities`	An array of strings where each string represents a quantity.
`dateTimes`	An array of strings where each string represents a DateTime (as it appears in the text) value.
`urls`	An array of strings where each string represents a URL
`emails`	An array of strings where each string represents an email
`namedEntities`	An array of complex types that contains the following fields: • category • value (The actual entity name) • offset (The location where it was found in the text) • confidence (Higher value means it's more to be a real entity)

`entities`	An array of complex types that contains rich information about the entities extracted from text, with the following fields
	• name (the actual entity name. This represents a "normalized" form)
	• wikipediaId
	• wikipediaLanguage
	• wikipediaUrl (a link to Wikipedia page for the entity)
	• bingId
	• type (the category of the entity recognized)
	• subType (available only for certain categories, this gives a more granular view of the entity type)
	• matches (a complex collection that contains)
	  ◦ text (the raw text for the entity)
	  ◦ offset (the location where it was found)
	  ◦ length (the length of the raw entity text)

Sample definition

JSON

```
{
  "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
  "categories": [ "Person", "Email"],
  "defaultLanguageCode": "en",
  "includeTypelessEntities": true,
  "minimumPrecision": 0.5,
```

```
    "inputs": [
      {
        "name": "text",
        "source": "/document/content"
      }
    ],
    "outputs": [
      {
        "name": "persons",
        "targetName": "people"
      },
      {
        "name": "emails",
        "targetName": "contact"
      },
      {
        "name": "entities"
      }
    ]
  }
```

Sample input

JSON

```
{
    "values": [
      {
        "recordId": "1",
        "data":
           {
             "text": "Contoso corporation was founded by John Smith. They
can be reached at contact@contoso.com",
             "languageCode": "en"
           }
      }
    ]
}
```

Sample output

JSON

```
{
  "values": [
    {
      "recordId": "1",
      "data" :
      {
```

```
        "persons": [ "John Smith"],
        "emails":["contact@contoso.com"],
        "namedEntities":
        [
          {
            "category":"Person",
            "value": "John Smith",
            "offset": 35,
            "confidence": 0.98
          }
        ],
        "entities":
        [
          {
            "name":"John Smith",
            "wikipediaId": null,
            "wikipediaLanguage": null,
            "wikipediaUrl": null,
            "bingId": null,
            "type": "Person",
            "subType": null,
            "matches": [{
                "text": "John Smith",
                "offset": 35,
                "length": 10
            }]
          },
          {
            "name": "contact@contoso.com",
            "wikipediaId": null,
            "wikipediaLanguage": null,
            "wikipediaUrl": null,
            "bingId": null,
            "type": "Email",
            "subType": null,
            "matches": [
            {
                "text": "contact@contoso.com",
                "offset": 70,
                "length": 19
            }]
          },
          {
            "name": "Contoso",
            "wikipediaId": "Contoso",
            "wikipediaLanguage": "en",
            "wikipediaUrl": "https://en.wikipedia.org/wiki/Contoso",
            "bingId": "349f014e-7a37-e619-0374-787ebb288113",
            "type": null,
            "subType": null,
            "matches": [
            {
                "text": "Contoso",
                "offset": 0,
                "length": 7
```

```
            }]
          }
        ]
      }
    }
  ]
}
```

Note that the offsets returned for entities in the output of this skill are directly returned from the Text Analytics API, which means if you are using them to index into the original string, you should use the StringInfo class in .NET in order to extract the correct content. More details can be found here.

Warning cases

If the language code for the document is unsupported, a warning is returned and no entities are extracted.

See also

- Built-in skills
- How to define a skillset
- Entity Recognition Skill (V3)

Additional resources

🕮 Documentation

Built-in skills - Azure Cognitive Search

Data extraction, natural language, and image processing skills add semantics and structure to raw content in an Azure Cognitive Search enrichment pipeline.

Key Phrase Extraction cognitive skill - Azure Cognitive Search

Evaluates unstructured text, and for each record, returns a list of key phrases in an AI enrichment pipeline in Azure Cognitive Search.

Custom skill interface - Azure Cognitive Search

Integrate a custom skill with an AI enrichment pipeline in Azure Cognitive Search through a web interface that defines compatible inputs and outputs in a skillset.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Conditional cognitive skill - Azure Cognitive Search

The conditional skill in Azure Cognitive Search enables filtering, creating defaults, and merging values in a skillset definition.

Text split cognitive skill - Azure Cognitive Search

Break text into chunks or pages of text based on length in an AI enrichment pipeline in Azure Cognitive Search.

Knowledge store concepts - Azure Cognitive Search

A knowledge store is enriched content created by an Azure Cognitive Search skillset and saved to Azure Storage for use in other apps and non-search scenarios.

Show 5 more

Sentiment cognitive skill (v2)

Article • 08/18/2022 • 2 minutes to read

The **Sentiment** skill (v2) evaluates unstructured text along a positive-negative continuum, and for each record, returns a numeric score between 0 and 1. Scores close to 1 indicate positive sentiment, and scores close to 0 indicate negative sentiment. This skill uses the machine learning models provided by Text Analytics in Cognitive Services.

> **Important**
>
> The Sentiment skill (v2) (**Microsoft.Skills.Text.SentimentSkill**) is now discontinued replaced by **Microsoft.Skills.Text.V3.SentimentSkill**. Follow the recommendations in **Deprecated cognitive search skills** to migrate to a supported skill.

> **Note**
>
> As you expand scope by increasing the frequency of processing, adding more documents, or adding more AI algorithms, you will need to **attach a billable Cognitive Services resource**. Charges accrue when calling APIs in Cognitive Services, and for image extraction as part of the document-cracking stage in Azure Cognitive Search. There are no charges for text extraction from documents.
>
> Execution of built-in skills is charged at the existing **Cognitive Services pay-as-you go price**. Image extraction pricing is described on the **Azure Cognitive Search pricing page**.

@odata.type

Microsoft.Skills.Text.SentimentSkill

Data limits

The maximum size of a record should be 5000 characters as measured by String.Length. If you need to break up your data before sending it to the sentiment analyzer, use the Text Split skill.

Skill parameters

Parameters are case-sensitive.

Parameter Name	Description
`defaultLanguageCode`	(optional) The language code to apply to documents that don't specify language explicitly. See the full list of supported languages.

Skill inputs

Input Name	Description
`text`	The text to be analyzed.
`languageCode`	(Optional) A string indicating the language of the records. If this parameter is not specified, the default value is "en". See the full list of supported languages.

Skill outputs

Output Name	Description
`score`	A value between 0 and 1 that represents the sentiment of the analyzed text. Values close to 0 have negative sentiment, close to 0.5 have neutral sentiment, and values close to 1 have positive sentiment.

Sample definition

JSON

```
{
    "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        },
        {
            "name": "languageCode",
            "source": "/document/languagecode"
        }
    ],
    "outputs": [
        {
            "name": "score",
```

```
            "targetName": "mySentiment"
        }
    ]
}
```

Sample input

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "text": "I had a terrible time at the hotel. The staff was
rude and the food was awful.",
                "languageCode": "en"
            }
        }
    ]
}
```

Sample output

JSON

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "score": 0.01
            }
        }
    ]
}
```

Warning cases

If your text is empty, a warning is generated and no sentiment score is returned. If a language is not supported, a warning is generated and no sentiment score is returned.

See also

- Built-in skills
- How to define a skillset
- Sentiment Skill (V3)

Demos - Azure Cognitive Search

Article • 03/15/2023 • 2 minutes to read

Demos are hosted apps that showcase search and AI enrichment functionality in Azure Cognitive Search. Several of these demos include source code on GitHub so that you can see how they were made.

The following demos are built and hosted by Microsoft.

Demo name	Description	Source code
AzSearchLab	A web front end that makes calls to a search index.	https://github.com/Azure-Samples/azure-search-lab
NYC Jobs demo	An ASP.NET app with facets, filters, details, geo-search (map controls).	https://github.com/Azure-Samples/search-dotnet-asp-net-mvc-jobs
JFK files demo	An ASP.NET web app built on a public data set, transformed with custom and predefined skills to extract searchable content from scanned document (JPEG) files. Learn more...	https://github.com/Microsoft/AzureSearch_JFK_Files
Semantic search for retail	Web app for a fictitious online retailer, "Terra"	Not available
Wolters Kluwer demo search app	Financial files demo that uses custom skills and forms recognition to make fictitious business documents searchable.	Not available

Additional resources

Documentation

Python samples - Azure Cognitive Search

Find Azure Cognitive Search demo Python code samples that use the Azure .NET SDK for Python or REST.

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Quickstart: Create a search index in Python - Azure Cognitive Search

Learn how to create a search index, load data, and run queries using Python, Jupyter Notebook, and the Azure.Documents.Search client library for Python.

What's new in Azure Cognitive Search

Announcements of new and enhanced features, including a service rename of Azure Search to Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Skillset concepts - Azure Cognitive Search

Skillsets are where you author an AI enrichment pipeline in Azure Cognitive Search. Learn important concepts and details about skillset composition.

Attach Cognitive Services to a skillset - Azure Cognitive Search

Learn how to attach a multi-service Cognitive Services resource to an AI enrichment pipeline in Azure Cognitive Search.

Incremental enrichment concepts (preview) - Azure Cognitive Search

Cache intermediate content and incremental changes from AI enrichment pipeline in Azure Storage to preserve investments in existing processed documents. This feature is currently in public preview.

Show 5 more

Productivity tools - Azure Cognitive Search

Article • 01/19/2023 • 2 minutes to read

Productivity tools are built by engineers at Microsoft, but aren't part of the Azure Cognitive Search service and aren't under Service Level Agreement (SLA). These tools are provided as source code that you can download, modify, and build to create an app that helps you develop or maintain a search solution.

Tool name	Description	Source code
Azure Cognitive Search Lab readme	Connects to your search service with a Web UI that exercises the full REST API, including the ability to edit a live search index.	https://github.com/Azure-Samples/azure-search-lab
Back up and Restore readme	Download a populated search index to your local device and then upload the index and its content to a new search service.	https://github.com/liamca/azure-search-backup-restore
Knowledge Mining Accelerator readme	Code and docs to jump start a knowledge store using your data.	https://github.com/Azure-Samples/azure-search-knowledge-mining
Performance testing readme	This solution helps you load test Azure Cognitive Search. It uses Apache JMeter as an open source load and performance testing tool and Terraform to dynamically provision and destroy the required infrastructure on Azure.	https://github.com/Azure-Samples/azure-search-performance-testing
Visual Studio Code extension	Although the extension is no longer available in the Visual Studio Code Marketplace, the code is open sourced at `https://github.com/microsoft/vscode-azurecognitivesearch`. You can clone and modify the tool for your own use.	

Additional resources

Training

Learning paths and modules

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Training - Azure Cognitive Search

Article • 10/12/2022 • 2 minutes to read

Training modules deliver an end-to-end experience that helps you build skills and develop insights while working through a progression of exercises. Visit the following links to begin learning with prepared lessons from Microsoft and other training providers.

- Introduction to Azure Cognitive Search (Microsoft)
- Implement knowledge mining with Azure Cognitive Search (Microsoft)
- Add search to apps (Pluralsight)
- Developer course (Pluralsight)

Additional resources

Documentation

Key Phrase Extraction cognitive skill - Azure Cognitive Search

Evaluates unstructured text, and for each record, returns a list of key phrases in an AI enrichment pipeline in Azure Cognitive Search.

Custom skill example using Bing Entity Search API - Azure Cognitive Search

Demonstrates using the Bing Entity Search service in a custom skill mapped to an AI-enriched indexing pipeline in Azure Cognitive Search.

Knowledge store concepts - Azure Cognitive Search

A knowledge store is enriched content created by an Azure Cognitive Search skillset and saved to Azure Storage for use in other apps and non-search scenarios.

Skillset concepts - Azure Cognitive Search

Skillsets are where you author an AI enrichment pipeline in Azure Cognitive Search. Learn important concepts and details about skillset composition.

Quickstart: Create a knowledge store in the Azure portal - Azure Cognitive Search

Use the Import data wizard to create a knowledge store used for persisting enriched content. Connect to a knowledge store for analysis from other apps, or send enriched content to downstream processes.

Custom Web API skill in skillsets - Azure Cognitive Search

Extend capabilities of Azure Cognitive Search skillsets by calling out to Web APIs. Use the Custom Web API skill to integrate your custom code.

Entity Recognition cognitive skill (v2) - Azure Cognitive Search

Extract different types of entities from text in an enrichment pipeline in Azure Cognitive Search.

Entity Recognition cognitive skill (v3) - Azure Cognitive Search

Extract different types of entities using the machine learning models of Azure Cognitive Services for Language in an AI enrichment pipeline in Azure Cognitive Search.

Show 5 more

Training

Learning path

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Certification

Microsoft Certified: Azure Developer Associate - Certifications

Azure developers design, build, test, and maintain cloud applications and services.

Partner spotlight

Article • 01/19/2023 • 3 minutes to read

Get expert help from Microsoft partners who build comprehensive solutions that include Azure Cognitive Search. The following partners have deep experience with integration of full-text search and AI enrichment across a range of business and technical scenarios.

Partner	Description	Product link
agolo	Agolo is the leading summarization engine for enterprise use. Agolo's AI platform analyzes hundreds of thousands of media articles, research documents and proprietary information to give each customer a summary of key points specific to their areas of interest. Our partnership with Microsoft combines the power and adaptability of the Azure Cognitive Search platform, integrated with Agolo summarization. Rather than typical search engine snippets, the results page displays contextually relevant Agolo summaries, instantly enabling the user to determine the relevance of that document to their specific needs. The impact of summarization-powered search is that users find more relevant content faster, enabling them to do their job more effectively and gaining a competitive advantage.	Product page
BA INSIGHT	BA Insight Search for Workplace is a complete enterprise search solution powered by Azure Cognitive Search. It is the first of its kind solution, bringing the internet to enterprises for secure, "askable", powerful search to help organizations get a return on information. It delivers a web-like search experience, connects to 80+ enterprise systems and provides automated and intelligent meta tagging.	Product page
BLUEGRANITE	BlueGranite offers 25 years of experience in Modern Business Intelligence, Data Platforms, and AI solutions across multiple industries. Their Knowledge Mining services enable organizations to obtain unique insights from structured and unstructured data sources. Modular AI capabilities perform searches on numerous file types to index data and associate that data with more traditional data sources. Analytics tools extract patterns and trends from the enriched data and showcase results to users at all levels.	Product page

Partner	Description	Product link
enlighten designs	Enlighten Designs is an award-winning innovation studio that has been enabling client value and delivering digitally transformative experiences for over 22 years. We are pushing the boundaries of the Microsoft technology toolbox, harnessing Cognitive Search, application development, and advanced Azure services that have the potential to transform our world. As experts in Power BI and data visualization, we hold the titles for the most viewed, and the most downloaded Power BI visuals in the world and are Microsoft's Data Journalism agency of record when it comes to data storytelling.	Product page
NEAL ANALYTICS	Neal Analytics offers over 10 years of cloud, data, and AI expertise on Azure. Its experts have recognized in-depth expertise across the Azure AI and ML services. Neal can help customers customize and implement Cognitive Search across a wide variety of use cases. Neal Analytics expertise ranges from enterprise-level search, form, and process automation to domain mapping for data extraction and analytics, plagiarism detection, and more.	Product page
NEUDESIC	Neudesic is the trusted technology partner in business innovation, delivering impactful business results to clients through digital modernization and evolution. Our consultants bring business and technology expertise together, offering a wide range of cloud and data-driven solutions, including custom application development, data and artificial intelligence, comprehensive managed services, and business software products. Founded in 2002, Neudesic is a privately held company headquartered in Irvine, California.	Product page
ORANGENXT	OrangeNXT offers expertise in data consolidation, data modeling, and building skillsets that include custom logic developed for specific use-cases. digitalNXT Search is an OrangeNXT solution that combines AI, optical character recognition (OCR), and natural language processing in Azure Cognitive Search pipeline to help you extract search results from multiple structured and unstructured data sources. Integral to digitalNXT Search is advanced custom cognitive skills for interpreting and correlating selected data.	Product page
plain concepts	Plain Concepts is a Microsoft Partner with over 15 years of cloud, data, and AI expertise on Azure, and more than 12 Microsoft MVP awards. We specialize in the creation of new data relationships among heterogeneous information sources, which combined with our experience with Artificial Intelligence, Machine Learning, and Cognitive Services, exponentially increases the productivity of both machines and human teams. We help customers to face the digital revolution with the AI-based solutions that best suits their company requirements.	Product page

Partner	Description	Product link
raytion	Raytion is an internationally operating IT business consultancy with a strategic focus on collaboration, search and cloud. Raytion offers intelligent and fully featured search solutions based on Microsoft Azure Cognitive Search and the Raytion product suite. Raytion's solutions enable an easy indexation of a broad range of enterprise content systems and provide a sophisticated search experience, which can be tailored to individual requirements. They are the foundation of enterprise search, knowledge searches, service desk agent support and many more applications.	Product page

Additional resources

Training

Learning paths and modules

Implement knowledge mining with Azure Cognitive Search - Training

Implement knowledge mining with Azure Cognitive Search

Learning certificate

Microsoft Certified: Azure AI Engineer Associate - Certifications

Azure AI engineers build, manage, and deploy AI solutions that leverage Azure Cognitive Services and Azure Applied AI services.

Terms of Use: Partner data sources

Article • 09/09/2022 • 2 minutes to read

The use of partner data source connectors may be subject to third-party terms and conditions. Microsoft isn't a party to your agreement(s) with third parties. Any concerns or questions about third-party terms or services should be directed to the third party. You're responsible for the indexing of data into your Azure tenant, as the data controller for your data.

Microsoft makes no representation, warranty, or guarantee in relation to the third-party data prior to indexing into your Azure tenant. Microsoft assumes no liability for errors that may occur during the indexing of third-party data. You agree that any data sourced from a third-party provider is provided "as is," and with respect to the index, is provided subject to your configuration. Indexed data in your Azure tenant will be considered "Customer Data" for purposes of the Online Services Terms, Privacy Statement, and Microsoft's other data obligations.

For more information on how we use Customer Data in Azure, see the Online Services Terms and Privacy Statement. You can also find out more about Microsoft's commitments to data protection and privacy by visiting our Trust Center.