

Contents

Azure Cognitive Search Documentation

Overview

[What is Cognitive Search?](#)

[Features in Cognitive Search](#)

[What's new in Cognitive Search](#)

Quickstarts

[Portal](#)

[Create a service](#)

[Create an index](#)

[Create a demo app](#)

[Create a skillset](#)

[Create a knowledge store](#)

[Query with Search explorer](#)

[ARM template](#)

[C#](#)

[Java](#)

[Node.js](#)

[Postman](#)

[PowerShell](#)

[Python](#)

Tutorials

[Create a C# app](#)

[1 - Basic search page](#)

[2 - Add results paging](#)

[3 - Add type-ahead](#)

[4 - Add facets](#)

[5 - Add results ordering](#)

[Index Azure data](#)

[Index Azure SQL Database](#)

- [Index Azure JSON blobs](#)
- [Index multiple Azure data sources](#)
- [Index any data](#)
- [Use AI to create content](#)
- [C#](#)
- [REST](#)
- [Python](#)
- [Debug a skillset](#)
- [Create a custom analyzer](#)
- [Query from Power Apps](#)
- [Samples](#)
 - [C# samples](#)
 - [Java samples](#)
 - [JavaScript samples](#)
 - [Python samples](#)
 - [REST samples](#)
- [Concepts](#)
 - [Full-text search](#)
 - [Indexers](#)
 - [AI enrichment](#)
 - [Skillsets](#)
 - [Debug sessions](#)
 - [Knowledge store](#)
 - [Projections](#)
 - [Incremental enrichment](#)
- [Security](#)
 - [Security overview](#)
 - [Security controls by Azure Policy](#)
 - [Securing indexer resources](#)
 - [Security baseline](#)
- [How-to guides](#)
 - [Create indexes and constructs](#)

[Create an index](#)

[Create a multi-language index](#)

[Analyzers](#)

[Adding analyzers](#)

[Add a language analyzer](#)

[Add a custom analyzer](#)

[Synonyms](#)

[Add synonyms](#)

[Synonyms example](#)

[Model complex data types](#)

[Model relational data](#)

[Import any data](#)

[Data import overview](#)

[Import data \(portal\)](#)

[Rebuild an index](#)

[Index large data sets](#)

[Handle concurrent updates](#)

[Import Azure data \(indexers\)](#)

[Work with indexers](#)

[Monitor indexers](#)

[Schedule indexers](#)

[Map fields](#)

[Connect with managed identities](#)

[Managed identities overview](#)

[Azure Storage](#)

[Azure Cosmos DB](#)

[SQL Database](#)

[Azure Blobs](#)

[Search over blobs](#)

[Understand blobs with AI](#)

[Configure a blob indexer](#)

[Index changed and deleted blobs](#)

- [Index one-to-many blobs](#)
- [Index plain text](#)
- [Index CSV blobs](#)
- [Index JSON blobs](#)
- [Index encrypted blobs](#)
- [Azure Tables](#)
- [Azure Cosmos DB](#)
- [Azure Data Lake Storage Gen2](#)
- [Azure SQL Databases](#)
- [Azure SQL Managed Instances](#)
- [Azure SQL Server VMs](#)
- [Use skills \(AI\)](#)
 - [Attach a Cognitive Services resource](#)
 - [Skillsets](#)
 - [Define a skillset](#)
 - [Reference an annotation](#)
 - [Map to index fields](#)
 - [Process image files](#)
 - [Cache \(incremental\) enrichment](#)
 - [Custom skills](#)
 - [Integrate custom skills](#)
 - [Example - Azure Functions \(Python\)](#)
 - [Example - Azure Functions](#)
 - [Example - Azure Form Recognizer](#)
 - [Example - Azure Machine Learning](#)
 - [Design tips](#)
 - [Knowledge store](#)
 - [Create using REST and Postman](#)
 - [View with Storage Explorer](#)
 - [Connect with Power BI](#)
 - [Projection examples](#)
 - [Query](#)

- [Query types and composition](#)
- [Create a simple query](#)
- [Use full Lucene syntax](#)
- [Partial terms and special characters](#)
- [Fuzzy search](#)
- [Autocomplete "Search-as-you-type"](#)
- [Create a suggester](#)
- [Add autocomplete and suggestions](#)
- [Syntax reference](#)
 - [Simple query syntax](#)
 - [Full Lucene query syntax](#)
 - [moreLikeThis \(preview\)](#)
- [Filter](#)
 - [Filter overview](#)
 - [Facet filters](#)
 - [Add faceted navigation](#)
 - [Troubleshoot collection filters](#)
 - [Understand collection filters](#)
 - [Filter by language](#)
- [Results](#)
 - [Work with results](#)
 - [Relevance scoring](#)
 - [Scoring profiles](#)
 - [Similarity algorithm](#)
- [Plan](#)
 - [Choose a tier](#)
 - [Service limits](#)
 - [Adjust capacity](#)
 - [Scale for performance](#)
 - [Multitenant architecture](#)
- [Develop](#)
 - [API versions](#)

[Preview feature list](#)

[Develop in .NET](#)

[Use Microsoft.Azure.Search \(v10\)](#)

[Get started \(v10\)](#)

[Upgrade the SDK](#)

[.NET SDK 11.0](#)

[.NET SDK 10.0](#)

[.NET SDK 9.0](#)

[.NET SDK 5.0](#)

[.NET SDK 3.0](#)

[.NET SDK 1.1](#)

[.NET Management SDK](#)

[Upgrade the REST API](#)

[Security](#)

[Data encryption](#)

[Customer-managed keys](#)

[Get key information](#)

[Inbound security](#)

[API access keys](#)

[Configure an IP firewall](#)

[Create a private endpoint](#)

[Identity-based access](#)

[Security filters](#)

[Filter on user identities](#)

[Outbound security \(indexers\)](#)

[Access over an IP range](#)

[Access through trusted service exception](#)

[Access through private endpoints](#)

[Manage](#)

[Portal](#)

[PowerShell](#)

[Move service across regions](#)

[Role-based admin access](#)

Monitor

[Fundamentals](#)

[Diagnostic logging](#)

[Visualize diagnostic logs](#)

[Monitor query activity](#)

[Search traffic analytics](#)

Troubleshoot

[Indexer issues](#)

[Indexer errors and warnings](#)

Reference

[Azure CLI](#)

[Azure PowerShell](#)

[Resource Manager template](#)

REST

[Search API \(Stable\)](#)

[Search API \(Preview\)](#)

[Management API \(Stable\)](#)

[Management API \(Preview\)](#)

.NET

[Search API](#)

[Management API](#)

Java

[Search API](#)

[Management API](#)

JavaScript

[Search API](#)

[Management API](#)

Python

[Search API](#)

[Management API](#)

OData language reference

[Overview](#)

[\\$filter](#)

[\\$orderby](#)

[\\$select](#)

[any, all](#)

[eq, ne, gt, lt, ge, le](#)

[search.ismatch, search.ismatchscoring](#)

[geo.distance, geo.intersects](#)

[and, or, not](#)

[search.in](#)

[search.score](#)

[OData Language Grammar](#)

[Skills reference](#)

[Built-in skills](#)

[Overview](#)

[Conditional](#)

[Custom Entity Lookup](#)

[Document Extraction](#)

[Entity Recognition](#)

[Image Analysis](#)

[Key Phrase Extraction](#)

[Language Detection](#)

[OCR](#)

[PII Detection](#)

[Sentiment](#)

[Shaper](#)

[Text Merger](#)

[Text Split](#)

[Text Translation](#)

[Custom skills](#)

[Azure Machine Learning \(AML\)](#)

[Custom Web API](#)

[Deprecated skills](#)

[Named Entity Recognition](#)

[Azure Policy built-ins](#)

[Resources](#)

[Azure Community Support](#)

[Azure Updates](#)

[Feedback Forum \(UserVoice\)](#)

[Compliance and certifications](#)

[Documentation links \(knowledge mining\)](#)

[Demo sites](#)

[Financial files demo](#)

[JFK files demo](#)

[Training](#)

[Introduction \(Microsoft Learn\)](#)

[Add search to apps \(Pluralsight\)](#)

[Developer course \(Pluralsight\)](#)

[FAQ](#)

[Pricing](#)

[Videos](#)

What is Azure Cognitive Search?

10/4/2020 • 7 minutes to read • [Edit Online](#)

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

When you create a Cognitive Search service, you get a search engine that performs indexing and query execution, persistent storage of indexes that you create and manage, and a query language for composing simple to complex queries. Optionally, a search service integrates 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 text analysis, or custom AI that you create in Azure Machine Learning or wrap inside Azure Functions.

Architecturally, a search service sits in between the external data stores that contain your un-indexed data, and a client app that sends query requests to a search index and handles the response. An index schema determines the structure of searchable content.

The two primary workloads of a search service are *indexing* and *querying*.

- Indexing brings text into to your search service and makes it searchable. Internally, inbound text is processed into tokens and stored in inverted indexes for fast scans. During indexing, you have the option of adding *cognitive skills*, either predefined ones from Microsoft or custom skills that you create. The subsequent analysis and transformations can result in new information and structures that did not previously exist, providing high utility for many search and knowledge mining scenarios.
- Once an index is populated with searchable data, your client app sends query requests to a search service and handles responses. All query execution is over a search index that you create, own, and store in your service. In your client app, the search experience is defined using APIs from Azure Cognitive Search, and can include relevance tuning, autocomplete, synonym matching, fuzzy matching, pattern matching, filter, and sort.

Functionality is exposed through a simple [REST API](#) or [.NET SDK](#) that masks the inherent complexity of information retrieval. You can also use the Azure portal for service administration and content management, with tools for prototyping and querying your indexes and skillsets. Because the service runs in the cloud, infrastructure and availability are managed by Microsoft.

When to use Cognitive Search

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

- Consolidation of heterogeneous content types into a private, user-defined search index. You can populate a search index with streams of JSON documents from any source. For supported sources on Azure, use an *indexer* to automate indexing. Control over the index schema and refresh schedule is a key reason for using Cognitive Search.
- Easy implementation of search-related features. Search APIs simplify query construction, faceted navigation, filters (including geo-spatial search), synonym mapping, autocomplete, and relevance tuning. Using built-in features, you can satisfy end-user expectations for a search experience similar to commercial web search engines.

- Raw content is large undifferentiated text or image files or application files stored in Azure Blob storage or Cosmos DB. You can apply [cognitive skills](#) during indexing to identify and extract text, create structure, or create new information such as translated text or entities.
- Content needs 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 use Cognitive Search

Step 1: Provision service

You can [create a free service](#) shared with other subscribers, or a [paid tier](#) that dedicates resources used only by your service. All quickstarts and tutorials can be completed on the free service.

For paid tiers, you can scale a service in two dimensions to calibrate resourcing based on production requirements:

- Add Replicas to grow your capacity to handle heavy query loads
- Add Partitions to grow storage for more documents

Step 2: Create an index

Define an index schema to map to reflect the structure of the documents you wish to search, similar to fields in a database. A search index is a specialized data structure that is optimized for fast query execution.

It's common to [create the index schema in the Azure portal](#), or programmatically using the [.NET SDK](#) or [REST API](#).

TIP

Start with the [Quickstart: Import data wizard](#) to create, load, and query an index in minutes.

Step 3: Load data

After you define an index, you're ready to upload content. You can use either a push or pull model.

The push model "pushes" JSON documents into an index using APIs from an [SDK](#) or [REST](#). The external dataset can be virtually any data source as long as the documents are JSON.

The pull model "pulls" data from sources on Azure and sends it to a search index. The pull model is implemented through [indexers](#) that streamline and automate aspects of data ingestion, such as connecting to, reading, and serializing data. Supported data sources include Azure Cosmos DB, Azure SQL, and Azure Storage.

Step 4: Send queries and handle responses

After populating an index, you can [issue search queries](#) to your service endpoint using simple HTTP requests with [REST API](#) or the [.NET SDK](#).

Step through [Create your first search app](#) to build and then extend a web page that collects user input and handles results. You can also use [Postman](#) for [interactive REST](#) calls or the built-in [Search Explorer](#) in Azure portal to query an existing index.

How it compares

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

COMPARED TO	KEY DIFFERENCES
Bing	<p>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.</p> <p>Azure Cognitive Search searches an index you define, populated with data and documents you own, often from diverse sources. Azure Cognitive Search has crawler capabilities for some data sources through indexers, but you can push any JSON document that conforms to your index schema into a single, consolidated searchable resource.</p>
Database search	<p>Many database platforms include a built-in search experience. SQL Server has full text search. 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.</p> <p>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 is not a replacement for true data storage and we don't recommend using it in that capacity. For more information, see this forum post.</p> <p>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.</p>

COMPARED TO	KEY DIFFERENCES
Dedicated search solution	<p>Assuming you have 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.</p> <p>A cloud service is the right choice if you want a turn-key solution with minimal overhead and maintenance, and adjustable scale.</p> <p>Within the cloud paradigm, several providers offer comparable baseline features, with full-text search, geo-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.</p>

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:

- Azure data integration (crawlers) at the indexing layer
- Azure portal for central management
- Azure scale, reliability, and world-class availability
- AI processing of raw data to make it more searchable, including text from images, or finding patterns in unstructured content.
- Linguistic and custom analysis, with analyzers for solid full text search in 56 languages
- [Core features common to search-centric apps](#): scoring, faceting, suggestions, synonyms, geo-search, and more.

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

Watch this video

In this 15-minute video, program manager Luis Cabrera introduces Azure Cognitive Search.

Features of Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

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 [What is Azure Cognitive Search?](#).

Indexing features

CATEGORY	FEATURES
Data sources	<p>Search indexes can accept text from any source, provided it is submitted as a JSON document.</p> <p>Indexers automate data ingestion from supported Azure data sources and handle JSON serialization. Connect to Azure SQL Database, Azure Cosmos DB, or Azure Blob storage to extract searchable content in primary data stores. Azure Blob indexers can perform <i>document cracking</i> to extract text from major file formats, including Microsoft Office, PDF, and HTML documents.</p>
Hierarchical and nested data structures	<p>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.</p>
Linguistic analysis	<p>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.</p> <p>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.</p> <p>Custom lexical analyzers are used for complex query forms such as phonetic matching and regular expressions.</p>

AI enrichment and knowledge mining

CATEGORY	FEATURES
----------	----------

CATEGORY	FEATURES
AI processing during indexing	AI enrichment for image and text analysis can be applied to an indexing pipeline to extract text information from raw content. A few examples of built-in skills include optical character recognition (making scanned JPEGs searchable), entity recognition (identifying an organization, name, or location), and key phrase recognition. You can also code custom skills to attach to the pipeline. You can also integrate Azure Machine Learning authored skills .
Storing enriched content for analysis and consumption in non-search scenarios	Knowledge store is an alternative output of an indexing pipeline. Instead of sending tokenized terms to an index, you can send enriched documents created by the indexing pipeline to a knowledge store, resident in either Azure Blob storage or Table storage, depending on the configuration. Knowledge stores are created from AI-based indexing (skillsets). The purpose of a knowledge store is to support downstream analysis or processing. With new information and structures in a knowledge store, you can attach it to a machine learning process or connect from Power BI to explore the data.
Cached content	Incremental enrichment (preview) limits processing to just the documents that are changed by specific edits to the pipeline, using cached content for the parts of the pipeline that do not change.

Query and user experience

CATEGORY	FEATURES
Free-form text search	<p>Full-text search is a primary use case for most search-based apps. Queries can be formulated using a supported syntax.</p> <p>Simple query syntax provides logical operators, phrase search operators, suffix operators, precedence operators.</p> <p>Full Lucene query syntax includes all operations in simple syntax, with extensions for fuzzy search, proximity search, term boosting, and regular expressions.</p>
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.
Geo-search	Azure Cognitive Search processes, filters, and displays geographic locations. It enables users to explore data based on the proximity of a search result to a physical location. Watch this video or review this sample to learn more.

CATEGORY	FEATURES
Filters and facets	<p>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).</p> <p>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.</p>
User experience	<p>Autocomplete can be enabled for type-ahead queries in a search bar.</p> <p>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.</p> <p>Synonyms associates equivalent terms that implicitly expand the scope of a query, without the user having to provide the alternate terms.</p> <p>Hit highlighting applies text formatting to a matching keyword in search results. You can choose which fields return highlighted snippets.</p> <p>Sorting is offered for multiple fields via the index schema and then toggled at query-time with a single search parameter.</p> <p>Paging and throttling your search results is straightforward with the finely tuned control that Azure Cognitive Search offers over your search results.</p>

Security features

CATEGORY	FEATURES
Data encryption	<p>Microsoft-managed encryption-at-rest is built into the internal storage layer and is irrevocable.</p> <p>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.</p>
Endpoint protection	<p>IP rules for inbound firewall support allows you to set up IP ranges over which the search service will accept requests.</p> <p>Create a private endpoint using Azure Private Link to force all requests through a virtual network.</p>

CATEGORY	FEATURES
Outbound security (indexers)	<p>Data access through private endpoints allows an indexer to connect to Azure resources that are protected through Azure Private Link.</p> <p>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.</p>

Portal features

CATEGORY	FEATURES
Tools for prototyping and inspection	<p>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.</p> <p>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.</p> <p>Search explorer is used to test queries and refine scoring profiles.</p> <p>Create demo app is used to generate an HTML page that can be used to test the search experience.</p>
Monitoring and diagnostics	<p>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 additional configuration required.</p>

Programmability

CATEGORY	FEATURES
REST	<p>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.</p> <p>Management REST API is for service creation and clean up through Azure Resource Manager. You can also use this API to manage keys and provision a service.</p>

CATEGORY	FEATURES
Azure SDK for .NET	<p>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.</p> <p>Microsoft.Azure.Management.Search is for service creation and clean up through Azure Resource Manager. You can also use this API to manage keys and provision a service.</p>
Azure SDK for Java	<p>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.</p> <p>com.microsoft.azure.management.search is for service creation and clean up through Azure Resource Manager. You can also use this API to manage keys and provision a service.</p>
Azure SDK for Python	<p>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.</p> <p>azure-mgmt-search is for service creation and clean up through Azure Resource Manager. You can also use this API to manage keys and provision a service.</p>
Azure SDK for JavaScript/TypeScript	<p>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.</p> <p>azure/arm-search is for service creation and clean up through Azure Resource Manager. You can also use this API to manage keys and provision a service.</p>

See also

- [What's new in Cognitive Search](#)
- [Preview features in Cognitive Search](#)

What's new in Azure Cognitive Search

10/4/2020 • 8 minutes to read • [Edit Online](#)

Learn what's new in the service. Bookmark this page to keep up to date with the service.

September 2020

Create an identity for a search service in Azure Active Directory, then use RBAC permissions to grant the identity read-only permissions to Azure data sources. Optionally, choose the [trusted service exception](#) capability if IP rules are not an option.

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Managed service identity	Indexers, security	<p>Create an identity for a search service in Azure Active Directory, then use RBAC permissions to grant access to Azure data sources. This approach eliminates the need for credentials on the connection string.</p> <p>An additional way to use a managed service identity is through a trusted service exception if IP rules are not an option.</p>	Generally available. Access this capability when using the portal or Create Data Source (REST) with api-version=2020-06-30.
Outbound requests using a private link	Indexers, security	<p>Create a shared private link resource that indexers can use when accessing Azure resources secured by Azure Private Link. For more information about all of the ways you can secure indexer connections, see Secure indexer resources using Azure network security features.</p>	Generally available. Access this capability when using the portal or Shared Private Link Resource with api-version=2020-08-01.
Management REST API (2020-08-01)	REST	New stable REST API adds support for creating shared private link resources.	Generally available.
Management REST API (2020-08-01-Preview)	REST	Adds shared private link resource for Azure Functions and Azure SQL for MySQL Databases.	Public preview.
Management .NET SDK 4.0	.NET SDK	Azure SDK update for the management SDK, targeted REST API version 2020-08-01.	Generally available.

August 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
double encryption	Security	Enable double encryption at the storage layer by configuring customer-managed key (CMK) encryption on new search services. Create a new service, configure and apply customer-managed keys to indexes or synonym maps, and benefit from double encryption over that content.	Generally available on all search services created after August 1, 2020 in these regions: West US 2, East US, South Central US, US Gov Virginia, US Gov Arizona. Use the portal, management REST APIs, or SDKs to create the service.

July 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Azure.Search.Documents client library	Azure SDK for .NET	<p>.NET client library released by the Azure SDK team, designed for consistency with other .NET client libraries.</p> <p>Version 11 targets the Search REST api-version=2020-06-30, but does not yet support knowledge store, geospatial types, or FieldBuilder.</p> <p>For more information, see Quickstart: Create an index and Upgrade to Azure.Search.Documents (v11).</p>	Generally available. Install the Azure.Search.Documents package from NuGet.
azure.search.documents client library	Azure SDK for Python	<p>Python client library released by the Azure SDK team, designed for consistency with other Python client libraries.</p> <p>Version 11 targets the Search REST api-version=2020-06-30.</p>	Generally available. Install the azure-search-documents package from PyPI.
@azure/search-documents client library	Azure SDK for JavaScript	<p>JavaScript client library released by the Azure SDK team, designed for consistency with other JavaScript client libraries.</p> <p>Version 11 targets the Search REST api-version=2020-06-30.</p>	Generally available. Install the @azure/search-documents package from npm.

June 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Knowledge store	AI enrichment	Output of an AI-enriched indexer, storing content in Azure Storage for use in other apps and processes.	Generally available. Use Search REST API 2020-06-30 or later, or the portal.
Search REST API 2020-06-30	REST	A new stable version of the REST APIs. In addition to knowledge store, this version includes enhancements to search relevance and scoring.	Generally available.
Okapi BM25 relevance algorithm	Query	New relevance ranking algorithm automatically used for all new search services created after July 15. For services created earlier, you can opt in by setting the <code>similarity</code> property on index fields.	Generally available. Use Search REST API 2020-06-30 or later, or REST API 2019-05-06.
<code>executionEnvironment</code>	Security (indexers)	Explicitly set this indexer configuration property to <code>private</code> to force all connections to external data sources over a private endpoint. Applicable only to search services that leverage Azure Private Link.	Generally available. Use Search REST API 2020-06-30 to set this general configuration parameter.

May 2020 (Microsoft Build)

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Debug sessions	AI enrichment	Debug sessions provide a portal-based interface to investigate and resolve issues with an existing skillset. Fixes created in the debug session can be saved to production skillsets. Get started with this tutorial .	Public preview, in the portal.
IP rules for in-bound firewall support	Security	Limit access to a search service endpoint to specific IP addresses.	Generally available. Use Management REST API 2020-03-13 or later, or the portal.

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Azure Private Link for a private search endpoint	Security	Shield a search service from the public internet by running it as a private link resource, accessible only to client apps and other Azure services on the same virtual network.	Generally available. Use Management REST API 2020-03-13 or later, or the portal.
system-managed identity (preview)	Security (indexers)	Register a search service as a trusted service with Azure Active Directory to set up connections to supported Azure data source for indexing. Applies to indexers that ingest content from Azure data sources such as Azure SQL Database, Azure Cosmos DB, and Azure Storage.	Public preview. Use the portal to register the search service.
sessionId query parameter, scoringStatistics=global parameter	Query (relevance)	Add sessionId to a query to establish a session for computing search scores, with scoringStatistics=global to collect scores from all shards, for more consistent search score calculations.	Generally available. Use Search REST API 2020-06-30 or later, or REST API 2019-05-06.
featuresMode (preview)	Query	Add this query parameter to expand a relevance score to show more detail: 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 algorithms. For a sample that demonstrates this capability, see Add machine learning (LearnToRank) to search relevance .	Public preview. Use Search REST API 2020-06-30-Preview or REST API 2019-05-06-Preview.

March 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Native blob soft delete (preview)	Indexers	An 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.	Public preview. Use the Search REST API 2020-06-30-Preview and REST API 2019-05-06-Preview, with Run Indexer against an Azure Blob data source that has native "soft delete" enabled.

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Management REST API (2020-03-13)	REST	New stable REST API for creating and managing a search service. Adds IP firewall and Private Link support	Generally available.

February 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
PII Detection (preview)	AI enrichment	A new cognitive skill used during indexing that extracts personal information from an input text and gives you the option to mask it from that text in various ways.	Public preview. Use the portal or Search REST API 2020-06-30-Preview or REST API 2019-05-06-Preview.
Custom Entity Lookup (preview)	AI enrichment	A new cognitive skill that looks for text from a custom, user-defined list of words and phrases. Using this list, it labels all documents with any matching entities. The skill also supports a degree of fuzzy matching that can be applied to find matches that are similar but not exact.	Public preview. Use the portal or Search REST API 2020-06-30-Preview or REST API 2019-05-06-Preview.

January 2020

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Customer-managed encryption keys	Security	Adds an extra layer of encryption in addition to the platform's built-in encryption. Using an encryption key that you create and manage, you can encrypt index content and synonym maps before the payload reaches a search service.	Generally available. Use Search REST API 2019-05-06 or later. For managed code, the correct package is still .NET SDK version 8.0-preview even though the feature is out of preview.
IP rules for in-bound firewall support (preview)	Security	Limit access to a search service endpoint to specific IP addresses. The preview API has new <code>IpRule</code> and <code>NetworkRuleSet</code> properties in CreateOrUpdate API . This preview feature is available in selected regions.	Public preview using <code>api-version=2019-10-01-Preview</code> .

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Azure Private Link for a private search endpoint (preview)	Security	Shield a search service from the public internet by running it as a private link resource, accessible only to client apps and other Azure services on the same virtual network.	Public preview using api-version=2019-10-01-Preview.

Features in 2019

December 2019

- [Create Demo App \(preview\)](#) is a new wizard in the portal that generates a downloadable HTML file with query (read-only) access to an index. The file comes with embedded script that renders an operational "localhost"-style web app, bound to an index on your search service. Pages are configurable in the wizard and can contain a search bar, results area, sidebar navigation, and typeahead query support. You can modify the HTML offline to extend or customize the workflow or appearance. A demo app is not easily extended to include security and hosting layers that are typically needed in production scenarios. You should consider it as a validation and testing tool rather than a short cut to a full client app.
- [Create a private endpoint for secure connections \(preview\)](#) explains how to set up a Private Link for secure connections to your search service. This preview feature is available upon request and uses [Azure Private Link](#) and [Azure Virtual Network](#) as part of the solution.

November 2019 - Ignite Conference

- [Incremental enrichment \(preview\)](#) adds caching and statefullness to an enrichment pipeline so that you can work on specific steps or phases without losing content that is already processed. Previously, any change to an enrichment pipeline required a full rebuild. With incremental enrichment, the output of costly analysis, especially image analysis, is preserved.
- [Document Extraction \(preview\)](#) is a cognitive skill used during indexing that allows you to extract the contents of a file from within a skillset. Previously, document cracking only occurred prior to skillset execution. With the addition of this skill, you can also perform this operation within skillset execution.
- [Text Translation](#) is a cognitive skill used during indexing that evaluates text and, for each record, returns the text translated to the specified target language.
- [Power BI templates](#) can jumpstart your visualizations and analysis of enriched content in a knowledge store in Power BI desktop. This template is designed for Azure table projections created through the [Import data wizard](#).
- [Azure Data Lake Storage Gen2 \(preview\)](#), [Cosmos DB Gremlin API \(preview\)](#), and [Cosmos DB Cassandra API \(preview\)](#) are now supported in indexers. You can sign up using [this form](#). You will receive a confirmation email once you have been accepted into the preview program.

July 2019

- Generally available in [Azure Government Cloud](#).

New service name

Azure Search is now renamed to **Azure Cognitive Search** to reflect the expanded (yet optional) use of cognitive skills and AI processing in core operations. API versions, NuGet packages, namespaces, and endpoints are unchanged. New and existing search solutions are unaffected by the service name change.

Service updates

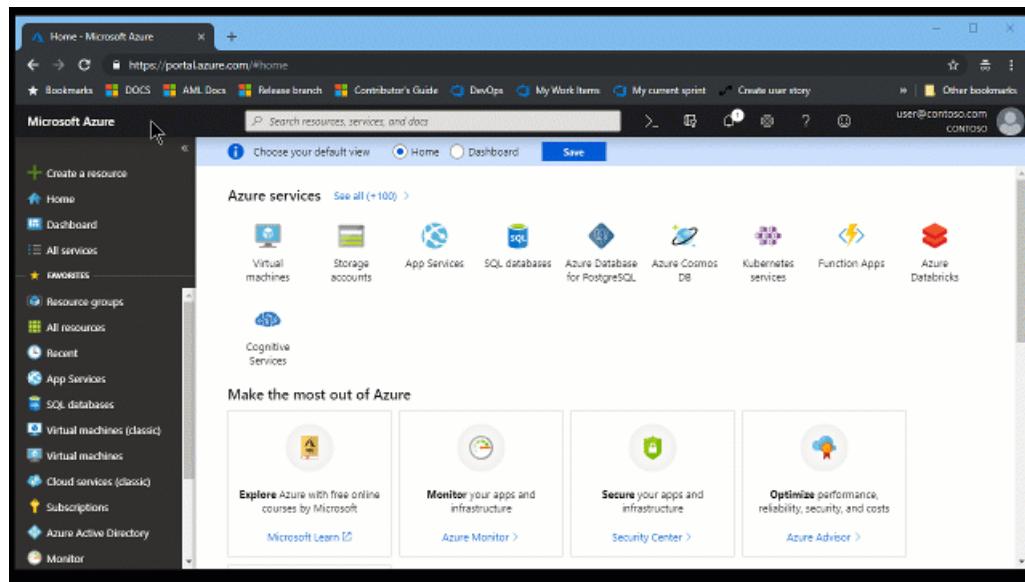
[Service update announcements](#) for Azure Cognitive Search can be found on the Azure web site.

Quickstart: Create an Azure Cognitive Search service in the portal

10/4/2020 • 8 minutes to read • [Edit Online](#)

Azure Cognitive Search is a standalone resource used to plug a search experience into custom apps. Cognitive Search integrates easily with other Azure services, with apps on network servers, or with software running on other cloud platforms.

In this article, learn how to create a resource in the [Azure portal](#).



Prefer PowerShell? Use the Azure Resource Manager [service template](#). For help with getting started, see [Manage Azure Cognitive Search with PowerShell](#).

Before you start

The following service properties are fixed for the lifetime of the service - changing any of them requires a new service. Because they are 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).
- Service tier [affects billing](#) and sets an upward limit on capacity. Some features are not available on the free tier.
- Service region can determine the availability of certain scenarios. If you need [high security features](#) or [AI enrichment](#), you will need to place Azure Cognitive Search in the same region as other services, or in regions that provide the feature in question.

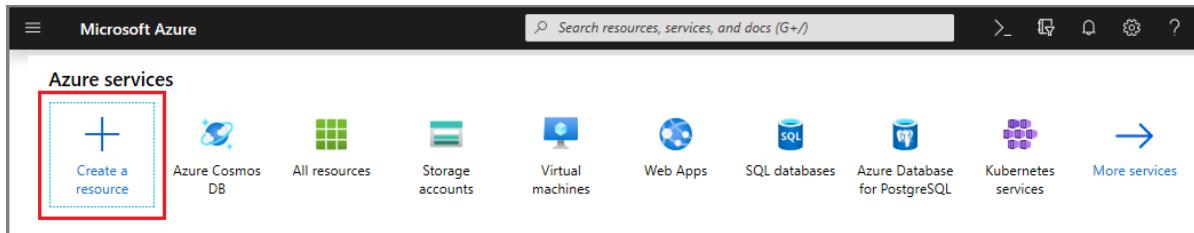
Subscribe (free or paid)

[Open a free Azure account](#) and use free credits to try out paid Azure services. After credits are used up, keep the account and continue to use free Azure services, such as Websites. Your credit card is never charged unless you explicitly change your settings and ask to be charged.

Alternatively, [activate MSDN subscriber benefits](#). An MSDN subscription gives you credits every month you can use for paid Azure services.

Find Azure Cognitive Search

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.



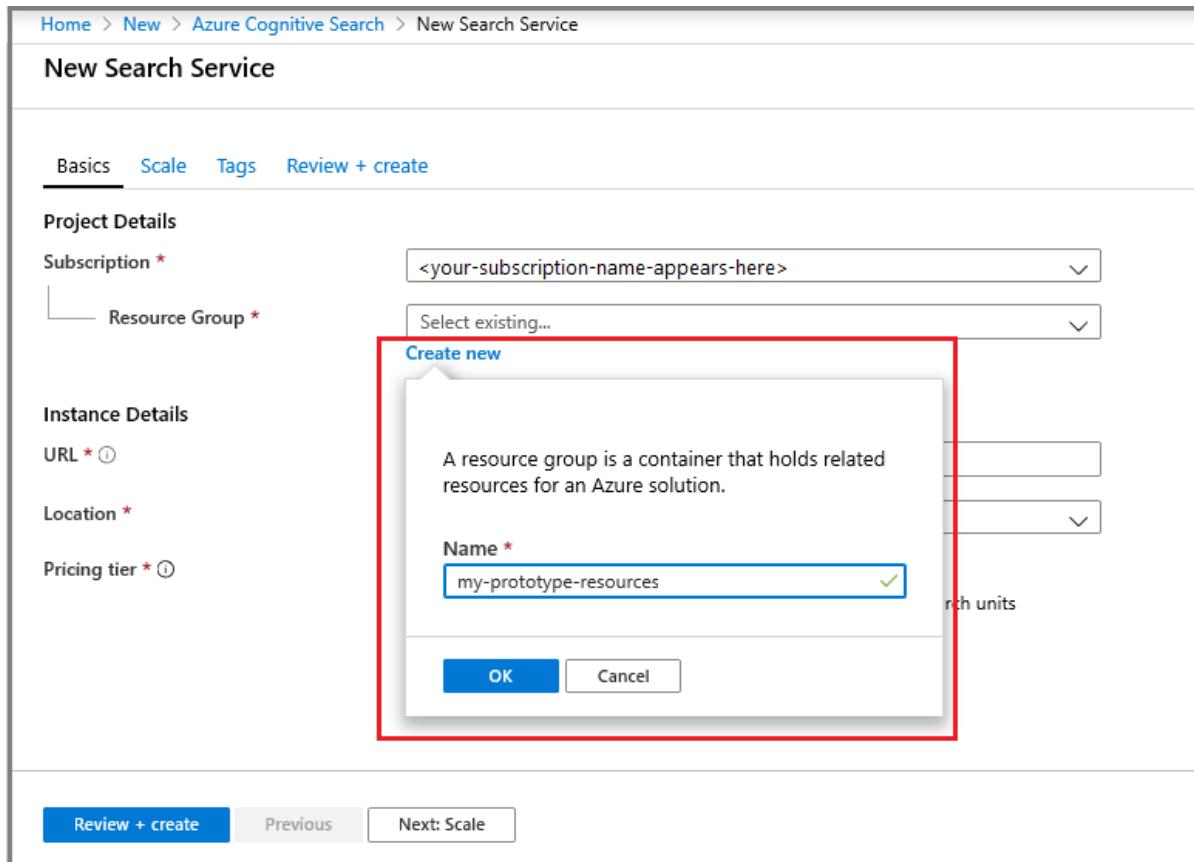
Choose a subscription

If you have more than one subscription, choose one for your search service. If you are implementing [double encryption](#) or other features that depend on managed service identities, choose the same subscription as the one used for Azure Key Vault or other services for which managed identities are used.

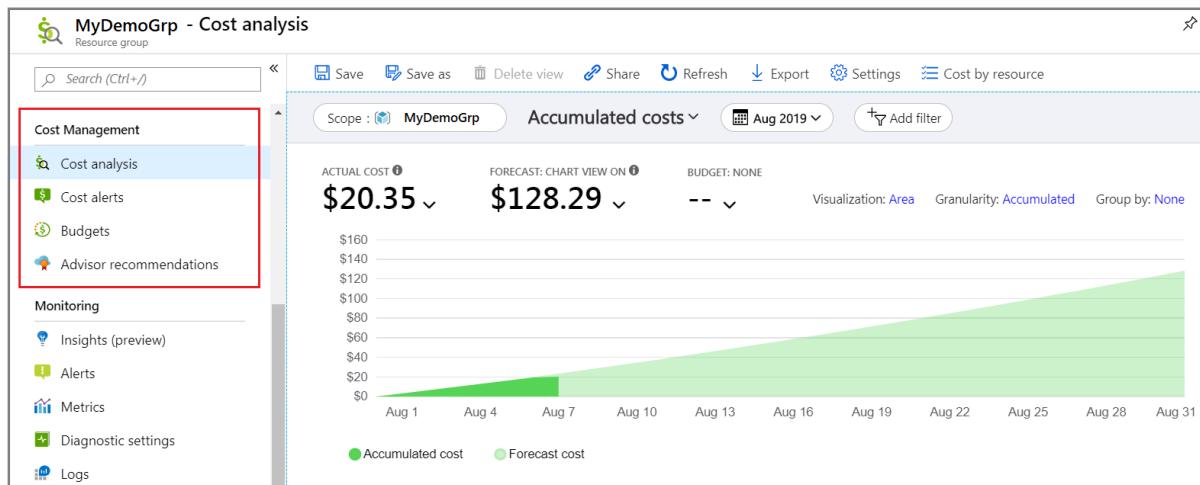
Set a resource group

A resource group is a container that holds related resources for your Azure solution. It is required for the search service. It is 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 are 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 ("")
- Do not 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 location

Azure Cognitive Search is available in most regions. The list of supported regions can be found in the [pricing page](#).

NOTE

Central India and UAE North are currently unavailable for new services. For services already in those regions, you can scale up with no restrictions, and your service is fully supported in that region. The restrictions are temporary and limited to new services only. We will remove this note when the restrictions no longer apply.

Double encryption is only available in certain regions. For more information, see [double encryption](#).

Requirements

If you are using AI enrichment, create your search service in the same region as Cognitive Services. *Co-location of Azure Cognitive Search and Cognitive Services in the same region is a requirement for AI enrichment.*

Customers with business continuity and disaster recovery (BCDR) requirements should create their services in [regional pairs](#). For example, if you are operating in North America, you might choose East US and West US, or North Central US and South Central US, for each service.

Recommendations

If you are using multiple Azure services, choose a region that is also hosting your data or application service. Doing so minimizes or voids bandwidth charges for outbound data (there are no charges for outbound data when services are in the same region).

Choose a pricing tier (SKU)

[Azure Cognitive Search is currently offered in multiple pricing tiers](#): Free, Basic, or Standard. Each tier has its own [capacity and limits](#). See [Choose a pricing tier or SKU](#) for guidance.

Basic and Standard are the most common choices for production workloads, but most customers start with the Free service. Key differences among tiers are partition size and speed, and limits on the number of objects you can create.

Remember, a pricing tier cannot be changed once the service is created. If you need a higher or lower tier, you will have to re-create the service.

Create your service

After you've provided the necessary inputs, go ahead and create the service.

Home > New > Azure Cognitive Search > New Search Service

New Search Service

Basics Scale Tags Review + create

Project Details

Subscription * <your-subscription-name-appears-here>

Resource Group * my-new-resource-group [Create new](#)

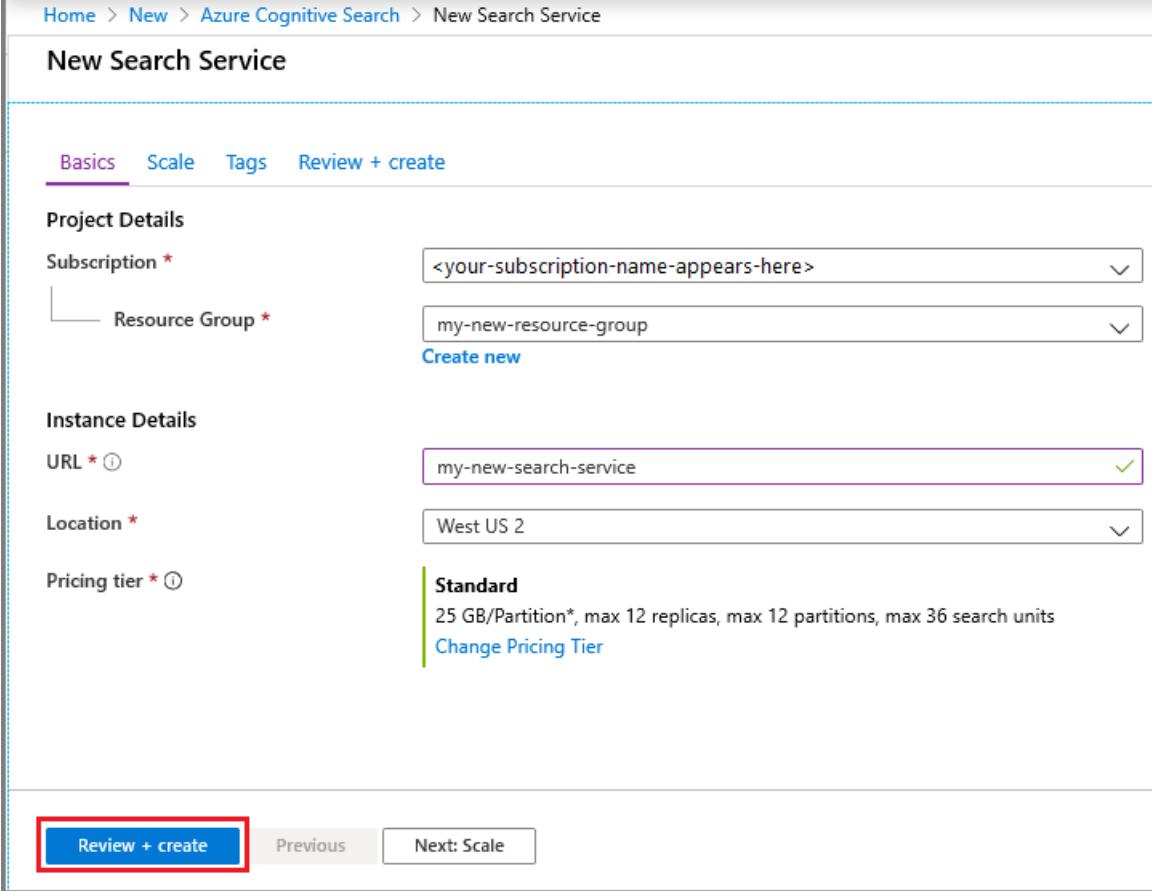
Instance Details

URL * my-new-search-service

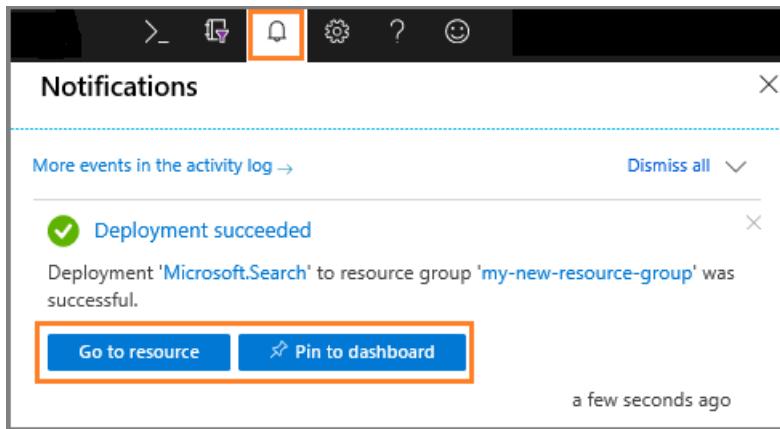
Location * West US 2

Pricing tier * Standard
25 GB/Partition*, max 12 replicas, max 12 partitions, max 36 search units
[Change Pricing Tier](#)

[Review + create](#) Previous Next: Scale



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.



Notifications

More events in the activity log → Dismiss all

Deployment succeeded Deployment 'Microsoft.Search' to resource group 'my-new-resource-group' was successful.

Go to resource Pin to dashboard a few seconds ago

Get a key and URL endpoint

Unless you are using the portal, programmatic access to your new service requires that you provide the URL endpoint and an authentication api-key.

1. On the **Overview** page, locate and copy the URL endpoint on the right side of the page.
2. On the **Keys** page, copy either one of the admin keys (they are equivalent). Admin api-keys are required for creating, updating, and deleting objects on your service. In contrast, query keys provide read-access to index content.

The screenshot shows two overlapping Azure portal pages. The top page is titled 'my-new-search-service' and has a red box around the 'Overview' tab in the left navigation. It displays a URL: <https://my-new-search-service.search.windows.net>. The bottom page is titled 'my-new-search-service - Keys' and also has a red box around the 'Keys' tab in its left navigation. A red circle labeled '2' highlights the 'Primary admin key' field, which contains the placeholder text '<placeholder-for-autogenerated-alphanumeric-string>'. A 'Copy to clipboard' button is visible next to this field.

An endpoint and key are not 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 your service is provisioned, you can scale it to meet your needs. If you chose the Standard tier for your Azure Cognitive Search service, you can scale your service in two dimensions: replicas and partitions. Had you chosen the Basic tier, you can only add replicas. If you provisioned the free service, scale is not 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 sidebar to add resources of either type.

The screenshot shows the 'Scale' configuration page for a search service. The 'Replicas' setting is set to 4, and the 'Partitions' setting is set to 3. The resulting value is 12 available search units. A note at the top states: 'Get 99.9% availability guaranteed with 3 replicas or more.'

NOTE

Per-partition storage and speed increases at higher tiers. For more information, see [capacity and limits](#).

When to add a second service

Most customers use just one service provisioned at a tier providing the [right balance of resources](#). 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 does not 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 is not 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](#).

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](#)

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

10/4/2020 • 12 minutes to read • [Edit Online](#)

Import data wizard is an Azure portal tool that guides you through the creation of a search index so that you can write interesting queries within minutes.

The wizard also has pages for AI enrichment so that you can extract text and structure from image files and unstructured text. Content processing with AI includes Optical Character Recognition (OCR), key phrase and entity extraction, and image analysis.

Prerequisites

Before you begin, you must have the following:

- 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.

Check for space

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

Sections on the service dashboard show how many indexes, indexers, and data sources you already have.

Home > Microsoft.Search - Overview > my-new-search-service

my-new-search-service
Search service

Search (Ctrl+ /) + Add index Import data Search explorer Refresh Delete Move

Get 99.9% availability guaranteed with 3 replicas or more. ↗

Resource group (change) : my-new-resource-group Url : https://my-new-search-service.search.windows.net

Status : Running Pricing tier : Standard

Location : West US 2 Replicas : 1 (No SLA)

Subscription (change) : Partitions : 1

Subscription ID : Search units : 1

Tags (change) : Click here to add tags

Usage Monitoring Indexes Data sources Skillsets

Name	Document Count	Storage Size
No indexes were found		

Create an index and load data

Search queries iterate over an [index](#) that contains searchable data, metadata, and additional constructs that optimize certain search behaviors.

For this tutorial, we use a built-in sample dataset that can be crawled using an [indexer](#) via the [Import data wizard](#). An indexer is a source-specific crawler that can read metadata and content from supported Azure data sources. Normally, indexers are used programmatically, but in the portal, you can access 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, click **Import data** on the command bar to create and populate a search index.



3. In the wizard, click **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.

Home > Microsoft.Search - Overview > my-new-search-service > Import data

Import data

[Connect to your data](#) Enrich content (Optional) Customize target index Create an indexer

Create and load a search index using data from an existing Azure data source in your current subscription. Azure Cognitive Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source	① Samples
Type	realstate-us-sample
	② hotels-sample

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

Typically, index creation is a code-based exercise, completed prior to loading data. However, as this tutorial indicates, the wizard can generate 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 data types and attributes. The check boxes across the top are *index attributes* controlling how the field is used.

- **Retrievable** means that it shows 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.
- **Key** is the unique document identifier. It's always a string, and it is required.
- **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 do not vary as a result of your selection. For example, if you set the **Retrievable** attribute on multiple fields, storage requirements do not go up.

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.

The screenshot shows the 'Import data' step of the Microsoft Search wizard. The 'Customize target index' tab is selected. The 'Index name' is set to 'hotels-sample-index'. The 'Key' is set to 'HotelId'. A 'Suggerer name' of 'sg' is specified, and the 'Search mode' is set to 'Match'. Below this, a table lists three fields: 'HotelId' (Edm.String), 'HotelName' (Edm.String), and 'Description' (Edm.String). The 'Retrievable' and 'Facetable' checkboxes are checked for all fields, while 'Filterable', 'Sortable', and 'Searchable' are unchecked. The 'Analyzer' dropdown shows 'English - Microsoft' for all fields, and the 'Suggerer' dropdown shows '...'. There are buttons for '+ Add field', '+ Add subfield', and 'Delete'.

2. Continue to the next page.

Step 4 - Configure indexer

Still in the **Import data** wizard, click **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.

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

Home > Microsoft.Search - Overview > my-new-search-service > Import data

Import data

Connect to your data Enrich content (Optional) Customize target index * Create an indexer *

Indexer

Name *

Schedule Once (highlighted with a red box)

Description (optional)

Advanced options

Previous: Customize target index **Submit** (highlighted with a red box and a red arrow pointing down to it)

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 click **Indexes**.

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.

Usage	Monitoring	Indexes	Data sources	Skillsets
Status	↑↓	Name	↑↓	Last run
In progress		hotels-sample-indexer		Just now

View the index

The main service page provides links to the resources created in your Azure Cognitive Search service. To view the index you just created, click **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.

Usage	Monitoring	Indexes	Indexes	Data sources	Skillsets
Name		Document Count		Storage Size	
hotels-sample-index		50		535.41 kB	

From this list, you can click 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. Scroll to the bottom of the list to enter a new field. In most cases, you

cannot change existing fields. Existing fields have a physical representation in Azure Cognitive Search and are thus non-modifiable, not even in code. To fundamentally change an existing field, create a new index, dropping the original.

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable	Analyzer	Suggester
HotelId	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	English - Microsoft	<input type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Microsoft	<input type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Microsoft	<input type="checkbox"/>

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

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

Query using Search explorer

Moving forward, you should now have a search index that's ready to query using the built-in [Search explorer](#) query page. It provides a search box so that you can test arbitrary query strings.

Search explorer is only equipped to handle [REST API requests](#), but it accepts syntax for both [simple query syntax](#) and [full Lucene query parser](#), plus all the search parameters available in [Search Document REST API](#) operations.

TIP

The following steps are demonstrated at 6m08s into the [Azure Cognitive Search Overview video](#).

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



2. From the Index dropdown, choose *hotels-sample-index*. Click the API Version dropdown, to see which REST APIs are available. For the queries below, use the generally available version (2020-06-30).

3. In the search bar, paste in the query strings below and click **Search**.

Example queries

You can enter terms and phrases, similar to what you might do in a Bing or Google search, or fully-specified query expressions. Results are returned as verbose JSON documents.

Simple query with top N results

Example (string query): `search=spa`

- The **search** parameter is used to input a keyword search for full text search, in this case, returning hotel data for those containing *spa* in any searchable field in the document.
- **Search explorer** returns results in JSON, which is verbose and hard to read if documents have a dense structure. This is intentional; visibility into the entire document is important for development purposes, especially during testing. For a better user experience, you will need to write code that [handles search results](#) to bring out important elements.
- Documents are composed of all fields marked as "retrievable" in the index. To view index attributes in the portal, click *hotels-sample* in the **Indexes** list.

Example (parameterized query): `search=spa&$count=true&$top=10`

- The **&** symbol is used to append search parameters, which can be specified in any order.
- The **\$count=true** parameter returns the total count of all documents returned. This value appears near the top of the search results. You can verify filter queries by monitoring changes reported by **\$count=true**. Smaller counts indicate your filter is working.
- The **\$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 via **\$top**.

Filter the query

Filters are included in search requests when you append the **\$filter** parameter.

Example (filtered): `search=beach&$filter=Rating gt 4`

- The **\$filter** parameter returns results matching the criteria you provided. In this case, ratings greater than 4.
- Filter syntax is an OData construction. For more information, see [Filter OData syntax](#).

Facet the query

Facet filters are included in search requests. You can use the facet parameter to return an aggregated count of documents that match a facet value you provide.

Example (faceted with scope reduction): `search=&facet=Category&$top=2`

- **search=*** is an empty search. Empty searches search over everything. One reason for submitting an empty query is to filter or facet over the complete set of documents. For example, you want a faceting navigation structure to consist of all hotels in the index.
- **facet** returns a navigation structure that you can pass to a UI control. It returns categories and a count. In this case, categories are based on a field conveniently called *Category*. There is no aggregation in Azure Cognitive Search, but you can approximate aggregation via **facet**, which gives a count of documents in each category.
- **\$top=2** brings back two documents, illustrating that you can use **top** to both reduce or increase results.

Example (facet on numeric values): `search=spa&facet=Rating`

- 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 the values it contains

(numeric, 1 through 5), are suitable for categorizing listings into groups.

- Only filterable fields can be faceted. Only retrievable fields can be returned in the results.
- The *Rating* field is double-precision floating point and the grouping will be by precise value. For more information on grouping by interval (for instance, "3 star ratings," "4 star ratings," etc.), see [How to implement faceted navigation in Azure Cognitive Search](#).

Highlight search results

Hit highlighting refers to formatting on text matching the keyword, given matches are found in a specific field. If your search term is deeply buried in a description, you can add hit highlighting to make it easier to spot.

Example (highlighter): `search=beach&highlight=Description`

- In this example, the formatted word *beach* is easier to spot in the description field.

Example (linguistic analysis): `search=beaches&highlight=Description`

- Full text search recognizes basic variations in word forms. In this case, search results contain highlighted text for "beach", for hotels that have that word in their searchable fields, in response to a keyword search on "beaches". Different forms of the same word can appear in results because of linguistic analysis.
- Azure Cognitive Search supports 56 analyzers from both Lucene and Microsoft. The default used by Azure Cognitive Search is the standard Lucene analyzer.

Try fuzzy search

By default, misspelled query terms, like *seatle* for "Seattle", fail to return matches in typical search. The following example returns no results.

Example (misspelled term, unhandled): `search=seatle`

To handle misspellings, you can use fuzzy search. Fuzzy search is enabled when you use the full Lucene query syntax, which occurs when you do two things: set **queryType=full** on the query, and append the ~ to the search string.

Example (misspelled term, handled): `search=seatle~&queryType=full`

This example now returns documents that include matches on "Seattle".

When **queryType** is unspecified, the default simple query parser is used. The simple query parser is faster, but if you require fuzzy search, regular expressions, proximity search, or other advanced query types, you will need the full syntax.

Fuzzy search and wildcard search have implications on search output. Linguistic analysis is not performed on these query formats. Before using fuzzy and wildcard search, review [How full text search works in Azure Cognitive Search](#) and look for the section about exceptions to lexical analysis.

For more information about query scenarios enabled by the full query parser, see [Lucene query syntax in Azure Cognitive Search](#).

Try geospatial search

Geospatial search is supported through the [edm.GeographyPoint data type](#) on a field containing coordinates. Geosearch is a type of filter, specified in [Filter OData syntax](#).

Example (geo-coordinate filters): `search=*&$count=true&$filter=geo.distance(Location,geography'POINT(-122.12 47.67)') le 5`

The example query filters all results for positional data, where results are less than 5 kilometers from a given point (specified as latitude and longitude coordinates). By adding **\$count**, you can see how many results are returned when you change either the distance or the coordinates.

Geospatial search is useful if your search application has a "find near me" feature or uses map navigation. It is not full text search, however. If you have user requirements for searching on a city or country/region by name, add fields containing city or country/region names, in addition to coordinates.

Takeaways

This tutorial 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 learned about [indexers](#), as well as the basic workflow for index design, including [supported modifications to a published index](#).

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 geo-search.

You also learned how to find indexes, indexers, and data sources in the portal. Given any new data source in the future, you can use the portal to quickly check its definitions or field collections with minimal effort.

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

Use a portal wizard to generate a ready-to-use web app that runs in a browser. You can try this wizard out 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)

10/4/2020 • 4 minutes to read • [Edit Online](#)

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 a remote index. A default app can include a search bar, results area, sidebar filters, and typeahead support.

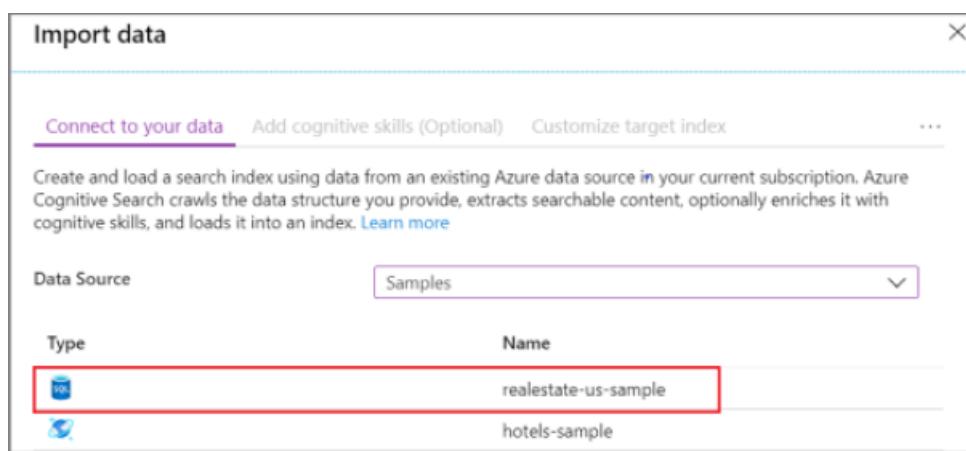
The demo app can help you visualize how an index will function in a client app, but it is not intended for production scenarios. Client apps should include security, error handling, and hosting logic that the generated HTML page doesn't provide. When you are ready to create a client app, see [Create your first search app using the .NET SDK](#) for next steps.

Prerequisites

Before you begin, you must have the following:

- 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

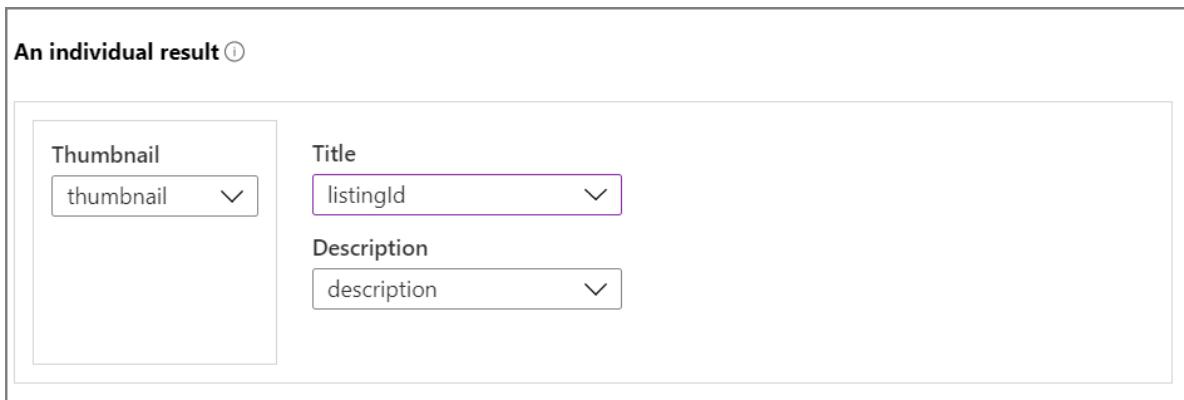
1. Sign in to the [Azure portal](#) with your Azure account.
2. [Find your search service](#) and on the Overview page, from the links on the middle of the page, select **Indexes**.

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.

Accept the current selection of facets 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.

The screenshot shows the 'Create Search App (preview)' interface. At the top, there are tabs for 'Individual result', 'Sidebar', and 'Suggestions'. The 'Suggestions' tab is active, showing a search bar with 'lake wa' and a dropdown menu with search results. Below the search bar, there's a section titled 'Suggestions of search box' with buttons for moving items up, down, to top, and to bottom. On the left, there's a list of styles: 'Normal', 'Inline with Previous Field' (repeated three times), and 'Inline with Previous Field' (with a dropdown arrow). On the right, there's a table for defining fields:

Field name	Show Field Name
number	<input checked="" type="checkbox"/>
street	<input checked="" type="checkbox"/>
city	<input checked="" type="checkbox"/>
region	<input checked="" type="checkbox"/>
countryCode	<input checked="" type="checkbox"/>

Create, download and execute

1. Select **Create demo app** to generate the HTML file.
2. When prompted, select **Download your app** to download the file.
3. Open the file. You should see a page 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 do not match the image. You can expect a more cohesive experience when you create an app based on your own indexes.

The screenshot shows a search application interface for 'Azure Search'. The search bar contains 'Lake Washington Boulevard South'. On the left, there are filter panels for 'type' (with 'House (249)' selected), 'price' (with a range from 0.0 to 69), and 'sqft' (with a range from 0.0 to 1.0m). The main area displays search results:

- 9383442**
This is a townhouse and is a short sale. This property has ocean views located close to a river and features a swimming pool, crown mouldings and a large walk in closet.
- 9382448**
This is a duplex residence and is brand new. This property has ocean views located close to schools and features a swimming pool, beautiful bedroom floors and vaulted ceilings.
- 9383199**
This is a multi-storey house and is a short sale. Enjoy water frontage located in a culs-de-sac and features top of the line appliances, french doors throughout and a covered front porch.

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

While the default app is useful for initial exploration and small tasks, reviewing the APIs early on will help you understand the concepts and workflow on a deeper level:

[Create an index using .NET SDK](#)

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

10/4/2020 • 8 minutes to read • [Edit Online](#)

A skillset is an AI-based feature that extracts information and structure from large undifferentiated text or image files, and makes the content both indexable and searchable in Azure Cognitive Search.

In this quickstart, you'll combine services and data in the Azure cloud to create the skillset. Once everything is in place, you'll run the **Import data** wizard in the Azure portal to pull it all together. The end result is a searchable index populated with data created by AI processing that you can query in the portal ([Search explorer](#)).

Prerequisites

Before you begin, you must have the following:

- 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.
- An Azure Storage account with [Blob storage](#).

NOTE

This quickstart also uses [Azure 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. This means that you can complete this exercise without having to create an additional 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. [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 if you want to try out the knowledge store feature later, in another walkthrough. Otherwise, choose any type.
3. Open the Blob services pages and create a container. You can use the default public access level.
4. In container, click **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 are not 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 Services and Content Intelligence.pptx	2018-04-02T21:10:50.000Z	Block Blob	application/vnd.openxmlformats-officedocument.presentationml.presentation
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-officedocument.wordprocessingml.document
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 are 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, click **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**, select the Storage account and container you created. Give the data source a name, and use default values for the rest.

Import data

Connect to your data * Add cognitive skills (Optional) Customize target index Create an indexer

Create and load a search index using data from an existing Azure data source in your current subscription. Azure Cognitive Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source	Azure Blob Storage
Name *	demo-skillset-ds
Data to extract	Content and metadata
Parsing mode	Default
Connection string *	DefaultEndpointsProtocol=https;AccountName=[accountName];AccountKe Choose an existing connection
Container name *	
Blob folder	your/folder/here

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 are 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.

Import data

Connect to your data * [Add cognitive skills \(Optional\)](#) [Customize target index](#) [Create an indexer](#)

i Enrich and extract structure from your documents through cognitive skills using the same AI algorithms that power Cognitive Services. Select the document cracking options and the cognitive skills you want to apply to your documents. Optionally, save enriched documents in Azure storage for use in scenarios other than search. [Learn more](#)

Attach Cognitive Services

To power your cognitive skills, select an existing Cognitive Services resource or create a new one. The Cognitive Services resource should be in the same region as your Azure Cognitive Search service.
The execution of cognitive skills will be billed to the selected resource. Otherwise, the number of enrichments executions will be limited. [Learn more](#)

[Refresh](#)

Cognitive Services Resource Name

↑↓ Region

↑↓

Free (Limited enrichments)

[Create new Cognitive Services resource](#)

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

Enable OCR to add image analysis skills to wizard page.

Set granularity to Pages to break up text into smaller chunks. Several text skills are limited to 5-KB inputs.

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

Import data

Add enrichments

Run cognitive skills over a source data field to create additional searchable fields. [Learn about additional skills and extensibility here](#).

Skillset name *

azureblob-skillset

Enable OCR and merge all text into **merged_content** field

Source data field *

merged_content

Enrichment granularity level

Pages (5000 characters chunks)

Text Cognitive Skills

Parameter

Field name

people

Extract people names

organizations

Extract organization names

locations

Extract location names

keyphrases

Extract key phrases

language

Detect language

Target Language English

translated_text

Translate text

Extract personally identifiable information

pii_entities

Detect sentiment

sentiment

Image Cognitive Skills

Field name

imageTags

Generate tags from images

imageCaption

Generate captions from images

imageCelebrities

Identify celebrities from images

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 properties for existing blobs plus new fields to contain 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.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACEABLE	SEARCHABLE	ANALYZER	SUGGESTER
content	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene	...
metadata_storage_content_type	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_size	Edm.Int64	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_last_modified	Edm.DateTime	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_name	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_path	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_content_encoding	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_content_type	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_language	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_title	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
people	Collection(Ed...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...
organizations	Collection(Ed...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...
locations	Collection(Ed...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...

Notice the strike-through and question mark on the **Retrievable** attribute by the `content` field. For text-heavy blob documents, the `content` field contains the bulk of the file, potentially running into thousands of lines. A field like this is unwieldy in search results and you should exclude it for this demo.

However, if you need to pass file contents to client code, make sure that **Retrievable** stays selected. Otherwise, consider clearing this attribute on `content` if the extracted elements (such as `people`, `organizations`, `locations`, and so forth) are sufficient.

Marking a field as **Retrievable** does not mean that the field *must* be present in the search results. You can precisely control search results composition by using the `$select` query parameter to specify which fields to include. For text-heavy fields like `content`, the `$select` parameter is your solution for providing manageable search results to the human users of your application, while ensuring client code has access to all the information it needs via the **Retrievable** attribute.

Continue to the next page.

Step 4 - Configure the indexer

The indexer is a high-level resource that drives the indexing process. It specifies the data source name, a target index, and frequency of execution. The **Import data** wizard creates several objects, and of them is always an indexer that you can run repeatedly.

- In the **Indexer** page, you can accept the default name and click the **Once** schedule option to run it immediately.

Import data

Connect to your data * Add cognitive skills (Optional) Customize target index * [Create an indexer *](#)

Indexer

Name *

Schedule ⓘ Once Hourly Daily Custom

Description

[Advanced options](#)

[Previous: Customize target index](#) **Submit**

2. Click **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 click **Indexes** in the middle of page.

Usage	Monitoring	Indexes	Data sources	Skillsets			
Status	↑↓	Name	↑↓	Last run	↑↓	Docs succeeded	↑↓
⚠ Warning		azureblob-indexer		Just now		14/14	

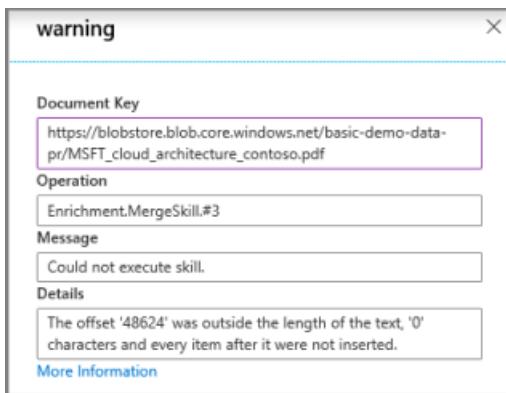
Warnings are normal given the wide range of content types. Some content types aren't valid for certain skills and on lower tiers it's common to encounter [indexer limits](#). For example, truncation notifications of 32,000 characters are an indexer limit on the Free tier. If you ran this demo on a higher tier, many truncation warnings would go away.

To check warnings or errors, click on the Warning status on the Indexers list to open the Execution History page.

On that page, click Warning status again to view the list of warnings similar to the one shown below.

Docs succeeded	14/14																																
Errors/Warnings	0/6																																
<input type="button"/> Search by Document Key, Operation, Message...																																	
<input type="button"/> Group By : Severity	<input type="button"/> Severity : All	<input type="button"/> Document Key : All	<input type="button"/> Operation : All	<input type="button"/> Message : All																													
<table border="1"> <thead> <tr> <th>Sev...</th> <th>Document Key</th> <th>Operation</th> <th>Message</th> </tr> </thead> <tbody> <tr> <td colspan="2">warning</td><td></td><td></td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>DocumentExtraction.azurebl...</td><td>Truncated extracted text to '32768' characters.</td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>DocumentExtraction.azurebl...</td><td>Truncated extracted text to '32768' characters.</td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>DocumentExtraction.azurebl...</td><td>Truncated extracted text to '32768' characters.</td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>DocumentExtraction.azurebl...</td><td>Truncated extracted text to '32768' characters.</td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>Enrichment.MergeSkill.#3</td><td>Could not execute skill.</td></tr> <tr> <td>⚠</td><td>https://blobstore.blob...</td><td>Enrichment.SplitSkill.#1</td><td>Could not execute skill because a skill input was invalid.</td></tr> </tbody> </table>		Sev...	Document Key	Operation	Message	warning				⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.	⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.	⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.	⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.	⚠	https://blobstore.blob...	Enrichment.MergeSkill.#3	Could not execute skill.	⚠	https://blobstore.blob...	Enrichment.SplitSkill.#1	Could not execute skill because a skill input was invalid.
Sev...	Document Key	Operation	Message																														
warning																																	
⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.																														
⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.																														
⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.																														
⚠	https://blobstore.blob...	DocumentExtraction.azurebl...	Truncated extracted text to '32768' characters.																														
⚠	https://blobstore.blob...	Enrichment.MergeSkill.#3	Could not execute skill.																														
⚠	https://blobstore.blob...	Enrichment.SplitSkill.#1	Could not execute skill because a skill input was invalid.																														

Details appear when you click a specific status line. This warning says that that merging stopped after reaching a maximum threshold (this particular PDF is large).



Query in Search explorer

After an index is created, you can run queries to return results. In the portal, use **Search explorer** for this task.

1. On the search service dashboard page, click **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=Microsoft&$select=people,organizations,locations,imageTags .
```

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

- Append `$select` to specify which fields to include 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.

The screenshot shows the Azure Cognitive Search search interface. At the top, there are tabs for 'Search explorer', 'Fields', 'CORS', 'Scoring profiles', and 'Index Definition (JSON)'. Below these are sections for 'Query string' (containing 'search=Microsoft&searchFields=organizations&\$select=people,organizations,locations,imageTags'), 'API version' (set to '2019-05-06'), and a 'Search' button. A 'Request URL' is also displayed. The main area is titled 'Results' and shows a JSON response. The response starts with a document ID '1' and contains an array of objects under 'value'. One object has a score of '1.2411621' and contains fields for 'people', 'organizations', 'locations', and 'imageTags'. The 'organizations' field contains an array with one item: 'Microsoft'. The 'imageTags' field contains an array of several tags: 'person', 'man', 'human face', 'clothing', 'shirt', 'indoor', and 'glasses'. The entire JSON response is as follows:

```
1 {  
2     "@odata.context": "https://mydemo.search.windows.net/indexes('azureblob-index2')/$metadata#docs(*)",  
3     "value": [  
4         {  
5             "@search.score": 1.2411621,  
6             "people": [],  
7             "organizations": [  
8                 "Microsoft"  
9             ],  
10            "locations": [],  
11            "imageTags": [  
12                "person",  
13                "man",  
14                "human face",  
15                "clothing",  
16                "shirt",  
17                "indoor",  
18                "glasses"  
19            ]  
20        },  
21    ]  
22}
```

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 is 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 filters](#), 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 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

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](#)

TIP

If you want to repeat this exercise or try a different AI enrichment walkthrough, delete the indexer in the portal. Deleting the indexer resets the free daily transaction counter back to zero for Cognitive Services processing.

Quickstart: Create an Azure Cognitive Search knowledge store in the Azure portal

10/4/2020 • 5 minutes to read • [Edit Online](#)

Knowledge store is a feature of Azure Cognitive Search that persists output from a content processing pipeline for subsequent analyses or downstream processing.

A pipeline accepts unstructured text and image content, applies AI powered by Cognitive Services (such as OCR and natural language processing), and outputs new structures and information that didn't previously exist. One of the physical artifacts created by a pipeline is a [knowledge store](#), which you can access through tools to analyze and explore content.

In this quickstart, you'll combine services and data in the Azure cloud to create a knowledge store. Once everything is in place, you'll run the **Import data** wizard in the portal to pull it all together. The end result is original text content plus AI-generated content that you can view in the portal ([Storage explorer](#)).

Prerequisites

Before you begin, you must have the following:

- 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.
- An Azure Storage account with [Blob storage](#).

NOTE

This quickstart also uses [Azure 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. This means that you can complete this exercise without having to create an additional 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 HotelReviews_Free.csv](#). This data is hotel review data saved in a CSV file (originates from Kaggle.com) and contains 19 pieces of customer feedback about a single hotel.
2. [Create an Azure storage account](#) or [find an existing account](#) under your current subscription. You'll use Azure storage for both the raw content to be imported, and the knowledge store that is the end result.
 - Choose the **StorageV2 (general purpose V2)** account type.
3. Open the Blob services pages and create a container named *hotel-reviews*.
4. Click **Upload**.



5. Select the **HotelReviews-Free.csv** file you downloaded in the first step.

Authentication method: Access key ([Switch to Azure AD User Account](#))
Location: hotelreviews-free

NAME	MODIFIED	ACCESS TIER	BLOB TYPE	SIZE	LEASE STATE
HotelReviews_Free.csv	7/29/2019, 11:57:58 AM	Hot (Infe...	Block blob	8.84 KiB	Available ...

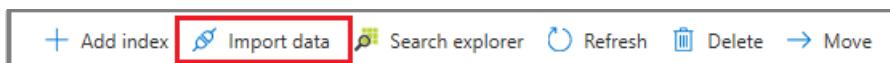
6. Before you quit the Blob storage pages, use a link on the left navigation pane to open the **Access Keys** page. Get a connection string to retrieve data from Blob storage. A connection string looks similar to the following example:

```
DefaultEndpointsProtocol=https;AccountName=<YOUR-ACCOUNT-NAME>;AccountKey=<YOUR-ACCOUNT-KEY>;EndpointSuffix=core.windows.net
```

You are 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, click **Import data** on the command bar to create a knowledge store in four steps.



Step 1: Create a data source

1. In **Connect to your data**, choose **Azure Blob storage**, select the account and container you created.
2. For the **Name**, enter `hotel-reviews-ds`.
3. For **Parsing mode**, select **Delimited text**, and then select the **First Line Contains Header** checkbox. Make sure the **Delimiter character** is a comma (,).
4. In **Connection String**, paste in the connection string you copied from the **Access Keys** page in Azure Storage.
5. In **Containers**, enter the name of the blob container holding the data.

Your page should look similar to the following screenshot.

Import data

* Connect to your data Add cognitive search (Optional) Customize target index Create an indexer

Create and load a search index using data from an existing Azure data source in your current subscription. Azure Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source	Azure Blob Storage
* Name	hotel-reviews-ds
Data to extract	Content and metadata
Parsing mode	Delimited text
First Line Contains Header	<input checked="" type="checkbox"/>
Delimiter Character	,
* Connection string	DefaultEndpointsProtocol=https;AccountName=mydemoblobstore;... Choose an existing connection
* Container name	hotel-reviews

Next: Add cognitive search (Optional)

6. Continue to the next page.

Step 2: Add cognitive skills

In this wizard step, you will create a skillset with cognitive skill enrichments. The source data consists of customer reviews in several languages. 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 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. Select these cognitive skills:
 - Extract key phrases
 - Translate text
 - Detect sentiment

^ Add enrichments

Run cognitive skills over a source data field to create additional searchable fields.
[Learn about additional skills and extensibility here.](#)

* Skillset name i
hotel-reviews-ss ✓

Enable OCR and merge all text into **merged_content** field i

* Source data field
reviews_text

Enrichment granularity level i
Pages (5000 characters chunks)

Text Cognitive Skills

	Field name
<input type="checkbox"/> Extract people names	people
<input type="checkbox"/> Extract organization names	organizations
<input type="checkbox"/> Extract location names	locations
<input checked="" type="checkbox"/> Extract key phrases	keyphrases
<input type="checkbox"/> Detect language	language
<input checked="" type="checkbox"/> Translate text	Target Language Eng... translated_text
<input checked="" type="checkbox"/> Detect sentiment	sentiment

7. Expand **Save enrichments to knowledge store**.

8. Select these **Azure table projections**:

- Documents
- Pages
- Key phrases

9. Enter the **Storage account Connection String** that you saved in a previous step.

Import data

- ✓ Attach Cognitive Services
- ✓ Add enrichments
- ^ Save enrichments to a knowledge store (Preview)

A knowledge store allows you to project your enriched documents into tables and blobs. [Learn more about knowledge store](#)

* Storage account connection string
DefaultEndpointsProtocol=https;AccountName=mydemoblobstore;AccountKey=UEX7p07/DfdorTEBUGfR... ✓

[Choose an existing connection](#)

Azure table projections

- Documents
- Pages
- Key phrases

Azure blob projections

- Document

[Previous: Connect to your data](#) [Next: Customize target index](#)

10. Optionally, download a Power BI template. When you access the template from the wizard, the local .pbix file is adapted to reflect the shape of your data.
11. Continue to the next page.

Step 3: Configure the index

In this wizard step, you will 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.

Import data

[Add field](#) [Add subfield](#) [Delete](#)

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACTETABLE	SEARCHABLE	ANALYZER
keyphrases	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce... ✓
translated_text	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Lucene ✓
sentiment	Collection(E...)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

[Previous: Add cognitive search \(Optional\)](#) [Next: Create an indexer](#)

3. Continue to the next page.

Step 4: Configure the indexer

In this wizard step, you will 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. Click **Submit** to run the indexer. Data extraction, indexing, application of cognitive skills all happen in this step.

Monitor status

Cognitive skill indexing takes longer to complete than typical text-based indexing. The wizard should open the Indexer list in the overview page so that you can track progress. For self-navigation, go to the Overview page and click **Indexers**.

In the Azure portal, you can also monitor the Notifications activity log for a clickable **Azure Cognitive Search notification** status link. Execution may take several minutes to complete.

Next steps

Now that you have enriched your data using Cognitive Services and projected the results into a knowledge store, you can use Storage Explorer or Power BI to explore your enriched data set.

You can view content in Storage Explorer, or take it a step further with Power BI to gain insights through visualization.

[View with Storage Explorer](#) [Connect with Power BI](#)

TIP

If you want to repeat this exercise or try a different AI enrichment walkthrough, delete the `hotel-reviews-idxr` indexer. Deleting the indexer resets the free daily transaction counter back to zero for Cognitive Services processing.

Quickstart: Use Search explorer to run queries in the portal

10/4/2020 • 4 minutes to read • [Edit Online](#)

Search explorer is a built-in query tool 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. Requests are formulated using the [Search REST API](#), with responses returned as JSON documents.

Prerequisites

Before you begin, you must have the following:

- 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. Use the [Import data](#) wizard to create the index. In the first step, when asked for the data source, choose **Samples** and then select the **realestate-us-sample** data source. Accept all of the wizard defaults to create the index.

Start Search explorer

1. In the [Azure portal](#), open the search service 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:

A screenshot of the Azure portal showing the 'realestate-us-sample-index' index page. The top navigation bar includes 'Home > mydemo > realestate-us-sample-index'. Below the title, there are 'Documents' (0) and 'Storage' (0 B). At the bottom of the page, there is a tab bar with 'Search explorer' (highlighted with a red box), 'Fields', 'CORS', 'Scoring profiles', and 'Index Definition (JSON)'. Underneath the tabs, there are sections for 'Query string' (with examples like *, \$top=10, \$top=10&\$skip=10&\$search=*) and 'API version' (set to 2019-05-06). A 'Search' button is also present. The 'Request URL' field shows the full search endpoint: https://mydemo.search.windows.net/indexes/realestate-us-sample-index/docs?api-version=2019-05-06&\$search=*.

Unspecified query

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 is 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=*`.

```
search=*
```

Results

Query string ⓘ
search=*

Request URL
[https://\[REDACTED\].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=*](https://[REDACTED].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=*)

Results

```
1 {
2   "@odata.context": "https://[REDACTED].search.windows.net/indexes('realestate-us-sample')/$metadata#docs(*)",
3   "value": [
4     {
5       "@search.score": 1,
6       "listingId": "9382104",
7       "beds": 3,
8       "baths": 3,
9       "description": "This is a multi-storey house and is priced to sell. This home provides coastal views located within walking distance of a beach and features a lap pool, heated towel racks and vaulted ceilings."}
```

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.

```
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 apartments "Lake Washington" miele OR thermador appliance

Request URL
[https://\[REDACTED\].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=Seattle%20apartments%20...](https://[REDACTED].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=Seattle%20apartments%20...)

Results

```
5   "@search.score": 0.69592, > thermador
6   "listingId": "9383988",
7   "beds": 2,
8   "baths": 2,
9   "description": "This is a flatlet but has a small mouse problem. This home provides lakefront property located in a culs-de-sac and features thermador appliances, wood floors and a guest house."}
```

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.

```
$count=true
```

Results

Query string ⓘ
`$count=true`

Request URL
`https://...search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=*&%24count=true`

Index: **realestate-us-sample**
API version: **2017-11-11**

Results

```
1  {
2    "@odata.context": "https://...search.windows.net/indexes
('realestate-us-sample')/$metadata#docs(*)",
3    "@odata.count": 4959,
4    "value": [
5      {
6        "@search.score": 1,
7        "listingId": "9382104",
8        "beds": 3,
9        "baths": 3,
```

Limit fields in search results

Add `$select` to limit results to the explicitly named fields for more readable output in **Search explorer**. To keep the search string and `$count=true`, prefix arguments with `&`.

```
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`

Request URL
`https://...search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=seattle%20condo&%24select...`

Index: **realestate-us-sample**
API version: **2017-11-11**

Results

```
1  {
2    "@odata.context": "https://...search.windows.net/indexes
('realestate-us-sample')/$metadata#docs(*)",
3    "@odata.count": 2056,
4    "value": [
5      {
6        "@search.score": 0.4378676,
7        "listingId": "9382100",
8        "beds": 5,
9        "baths": 5,
10       "description": "This is a condo and is brand new. This property has lake
access located close to schools and features a steam room, crown mouldings and a guest
room.",
11       "street": "23rd Avenue South",
12       "city": "Seattle",
13       "price": 4665600
```

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. 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.

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

Results

Query string ⓘ
`islect=listingId,beds,baths,description,street,city,price&$count=true&$top=100&$skip=50` \$top=100&\$skip=50 Search Index: **realestate-us-sample**
API version: **2017-11-11**

Request URL
`https://[REDACTED].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=seattle%20condo&%24select...` Copy

Results

```
1 {  
2   "@odata.context": "https://[REDACTED].search.windows.net/indexes  
('realestate-us-sample')/$metadata#docs(*)",  
3   "@odata.count": 2056,  
4   "value": [  
5     {  
6       "@search.score": 0.38382116,  
7       "listingId": "9383937",  
8       "beds": 1,  
9       "baths": 1,
```

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:

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

Results

Query string ⓘ
`search=seattle condo&$filter=beds gt 3&$count=true` Search Index: **realestate-us-sample**
API version: **2017-11-11**

Request URL
`https://[REDACTED].search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=seattle%20condo&%24filter=...` Copy

Results

```
1 {  
2   "@odata.context": "https://[REDACTED].search.windows.net/indexes  
('realestate-us-sample')/$metadata#docs(*)",  
3   "@odata.count": 805,  
4   "value": [  
5     {  
6       "@search.score": 0.4378676,  
7       "listingId": "9382100",  
8       "beds": 5,  
9       "baths": 5,  
10      "description": "This is a condo and is brand new. This property has lake  
access located close to schools and features a steam room, crown mouldings and a guest  
room.",
```

Order-by expressions

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:

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

Results

The screenshot shows the Azure Cognitive Search Search Explorer interface. At the top, there is a query string input field containing: `e condo&$select=listId,beds,price&$filter=beds gt 3&$count=true&$orderby=price asc`. To the right of the input field are a "Search" button and status information: "Index: realestate-us-sample" and "API version: 2017-11-11". Below the input field is a "Request URL" field showing the full URL: `https://search.windows.net/indexes/realestate-us-sample/docs?api-version=2017-11-11&search=seattle%20condo&%24select...`. A blue "Copy" button is to the right of the URL. The main area is titled "Results" and displays a JSON array of search results. The results are numbered 11 through 22. Each result object contains fields: `@search.score`, `listingId`, `beds`, and `price`. The `price` field values for the first two results (11 and 12) are highlighted with red boxes: 518400 and 520992 respectively.

11	{	<code>"@search.score": 0.05322655,</code>
12		<code>"listingId": "9385214",</code>
13		<code>"beds": 4,</code>
14		<code>"price": 518400</code>
15	,	
16	{	
17		<code>"@search.score": 0.05157484,</code>
18		<code>"listingId": "9382920",</code>
19		<code>"beds": 4,</code>
20		<code>"price": 520992</code>
21	,	
22	}	

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. You can use query expressions, shown in the examples, to limit which fields are returned.
- Documents are composed of all fields marked as **Retrievable** in the index. To view index attributes in the portal, click *realestate-us-sample* in the **Indexes** list on the search overview page.
- Free-form queries, 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 realestate 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 supported by Azure Cognitive Search. The default is a [simple syntax](#), but you can optionally use [full Lucene](#) for more powerful queries. [Filter expressions](#) are 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 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

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

[Create a basic query in Postman](#)

Quickstart: Deploy Cognitive Search using an ARM template

10/4/2020 • 3 minutes to read • [Edit Online](#)

This article walks you through the process for using an Azure Resource Manager template (ARM template) to deploy an Azure Cognitive Search resource in the Azure portal.

An [ARM template](#) is a JavaScript Object Notation (JSON) file that defines the infrastructure and configuration for your project. The template uses declarative syntax, which lets you state what you intend to deploy without having to write the sequence of programming commands to create it.

If 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.



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](#).

```
{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "name": {
      "type": "string",
      "minLength": 2,
      "maxLength": 60,
      "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",
      "allowedValues": [
        "free",
        "basic",
        "standard",
        "standard2",
        "standard3",
        "storage_optimized_11",
        "storage_optimized_12"
      ],
      "metadata": {
        "description": "The pricing tier of the search service you want to create (for example, basic or standard)."
      }
    },
    "replicaCount": {
      "type": "int"
    }
  }
}
```

```

        "type": "int",
        "defaultValue": 1,
        "minValue": 1,
        "maxValue": 12,
        "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": "2020-03-13",
    "name": "[parameters('name')]",
    "location": "[parameters('location')]",
    "sku": {
        "name": "[toLowerCase(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.



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.

Dashboard > Azure Search service

Azure Search service

Azure quickstart template

TEMPLATE

101-azure-search-create
1 resource

Edit template Edit paramet... Learn more

BASICS

Subscription *

Resource group *
Create new

Location *

SETTINGS

Name *

Sku standard

Replica Count 1

Partition Count 1

Hosting Mode default

Location [resourceGroup().location]

TERMS AND CONDITIONS

Template information | Azure Marketplace Terms | Azure Marketplace

By clicking "Purchase," I (a) agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

I agree to the terms and conditions stated above

Purchase

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 a search index using the Azure.Search.Documents client library

10/4/2020 • 9 minutes to read • [Edit Online](#)

Use the new [Azure.Search.Documents \(version 11\) client library](#) to create a .NET Core console application in C# that creates, loads, and queries a search index.

[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

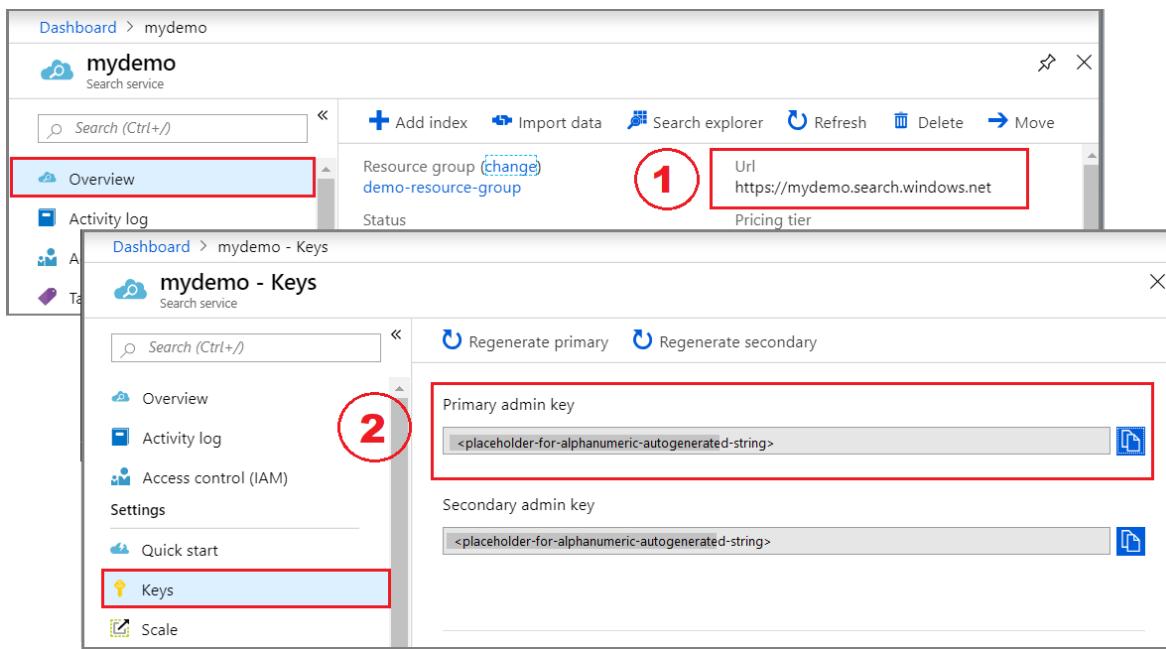
Before you begin, have the following tools and services:

- 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.

Get a key and endpoint

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, required if you are 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.

Set up your project

Start Visual Studio and create a new Console App project that can run on .NET Core.

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. In Tools > NuGet Package Manager, select Manage NuGet Packages for Solution....
2. Click Browse.
3. Search for `Azure.Search.Documents` and select version 11.0.0.
4. Click 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.

```
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` works with an existing index. Both need the service endpoint and an admin API key for authentication with create/delete rights.

```

static void Main(string[] args)
{
    string serviceName = "<YOUR-SERVICE-NAME>";
    string indexName = "hotels-quickstart-v11";
    string apiKey = "<YOUR-ADMIN-API-KEY>";

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

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

```

1 - Create an index

This quickstart builds a Hotels index that you'll load with hotel data and run queries on. 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. In `Hotel.cs`, define the structure of a hotel document.

```

using System;
using System.Text.Json.Serialization;

namespace AzureSearch.SDK.Quickstart.v11
{
    public class Hotel
    {
        [JsonPropertyName("hotelId")]
        public string Id { get; set; }

        [JsonPropertyName("hotelName")]
        public string Name { get; set; }

        [JsonPropertyName("hotelCategory")]
        public string Category { get; set; }

        [JsonPropertyName("baseRate")]
        public Int32 Rate { get; set; }

        [JsonPropertyName("lastRenovationDate")]
        public DateTime Updated { get; set; }
    }
}

```

3. In `Program.cs`, specify the fields and attributes. `SearchIndex` and `CreateIndex` are used to create an index.

```

// Define an index schema using SearchIndex
// Create the index using SearchIndexClient
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 }
    }
};

Console.WriteLine("{0}", "Creating index...\n");
idxclient.CreateIndex(index);

```

Attributes on the field determine how it is used in an application. For example, the `IsFilterable` attribute must be assigned to every field that supports a filter expression.

In contrast with previous versions of the .NET SDK that require `IsSearchable` on searchable string fields, you can use `SearchableField` and `SimpleField` to streamline field definitions.

Similar to the previous versions, other attributes are still required on the definition itself. For example, `IsFilterable`, `IsSortable`, and `IsFacetable` must be explicitly attributed, as shown in the sample above.

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, 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 five hotels in the code itself.

When uploading documents, you must use an `IndexDocumentsBatch` object. An `IndexDocumentsBatch` 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

`ndexDocumentsBatch` The documents below conform to the `hotels-quickstart-v11` index, as defined by the `hotel` class.

```

// Load documents (using a subset of fields for brevity)
IndexDocumentsBatch<Hotel> batch = IndexDocumentsBatch.Create(
    IndexDocumentsAction.Upload(new Hotel { Id = "78", Name = "Upload Inn", Category = "hotel", Rate
= 279, Updated = new DateTime(2018, 3, 1, 7, 0, 0 ) }),
    IndexDocumentsAction.Upload(new Hotel { Id = "54", Name = "Breakpoint by the Sea", Category =
"motel", Rate = 162, Updated = new DateTime(2015, 9, 12, 7, 0, 0 ) }),
    IndexDocumentsAction.Upload(new Hotel { Id = "39", Name = "Debug Motel", Category = "motel", Rate
= 159, Updated = new DateTime(2016, 11, 11, 7, 0, 0 ) }),
    IndexDocumentsAction.Upload(new Hotel { Id = "48", Name = "NuGet Hotel", Category = "hotel", Rate
= 238, Updated = new DateTime(2016, 5, 30, 7, 0, 0 ) }),
    IndexDocumentsAction.Upload(new Hotel { Id = "12", Name = "Renovated Ranch", Category = "motel",
Rate = 149, Updated = new DateTime(2020, 1, 24, 7, 0, 0 ) }));

IndexDocumentsOptions idxoptions = new IndexDocumentsOptions { ThrowOnAnyError = true };

Console.WriteLine("{0}", "Loading index...\n");
qryclient.IndexDocuments(batch, idxoptions);

```

Once you initialize the [IndexDocumentsBatch](#) object, you can send it to the index by calling [IndexDocuments](#) on your [SearchClient](#) object.

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

```

// 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) as well as other [options](#).

The [SearchResults](#) class represents the results.

- In [Program.cs](#), create a [WriteDocuments](#) method that prints search results to the console.

```

private static void WriteDocuments(SearchResults<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> response in searchResults.GetResults())
    {
        Hotel doc = response.Document;
        var score = response.Score;
        Console.WriteLine($"Name: {doc.Name}, Type: {doc.Category}, Rate: {doc.Rate}, Last-update:
{doc.Updated}, Score: {score}");
    }

    Console.WriteLine();
}

```

- Create a [RunQueries](#) method to execute queries and return results. Results are Hotel objects.

```

private static void RunQueries(SearchClient qryclient)
{
    SearchOptions options;
    SearchResults<Hotel> response;

    Console.WriteLine("Query #1: Search on the term 'motel' and list the relevance score for each match...\n");

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

    response = qryclient.Search<Hotel>("motel", options);
    WriteDocuments(response);

    Console.WriteLine("Query #2: Find hotels where 'type' equals hotel...\n");

    options = new SearchOptions()
    {
        Filter = "hotelCategory eq 'hotel'",
    };

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

    Console.WriteLine("Query #3: Filter on rates less than $200 and sort by when the hotel was last updated...\n");

    options = new SearchOptions()
    {
        Filter = "baseRate lt 200",
        OrderBy = { "lastRenovationDate desc" }
    };

    response = qryclient.Search<Hotel>("*", options);
    WriteDocuments(response);
}

```

This example shows the two [ways of matching terms in a query](#): full-text search, and filters:

- Full-text search queries for one or more terms in searchable fields in your index. The first query is full text search. Full-text search produces relevance scores used to rank the results.
- Filter is a boolean expression that is evaluated over [IsFilterable](#) fields in an index. Filter queries either include or exclude values. As such, there is no relevance score associated with a filter query. The last two queries demonstrate filter search.

You can use full-text search and filters together or separately.

Both searches 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 do not 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 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 C# quickstart, you worked through a series 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. If you are comfortable with the basic concepts, we recommend the next article for an exploration of alternative approaches and concepts that will deepen your knowledge.

[How to develop in .NET](#)

Want to optimize and save on your cloud spending?

[Start analyzing costs with Cost Management](#)

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

10/4/2020 • 18 minutes to read • [Edit Online](#)

Create a Java console application that creates, loads, and queries a search index using [IntelliJ, Java 11 SDK](#), and the [Azure Cognitive Search REST API](#). 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:

- [IntelliJ IDEA](#)
- [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.

Create a query key, too. It's a best practice to issue query requests with read-only access.

The screenshot shows the Microsoft Azure portal interface for managing keys for a search service named "mydemo". The "Keys" section is selected in the left sidebar. The primary admin key is set to a placeholder. A secondary admin key is present but empty. A table lists a single query key with both its name and key values set to placeholders.

NAME	KEY
<placeholder-for-alphanumeric-autogenerated-string>	<placeholder-for-alphanumeric-autogenerated-string>

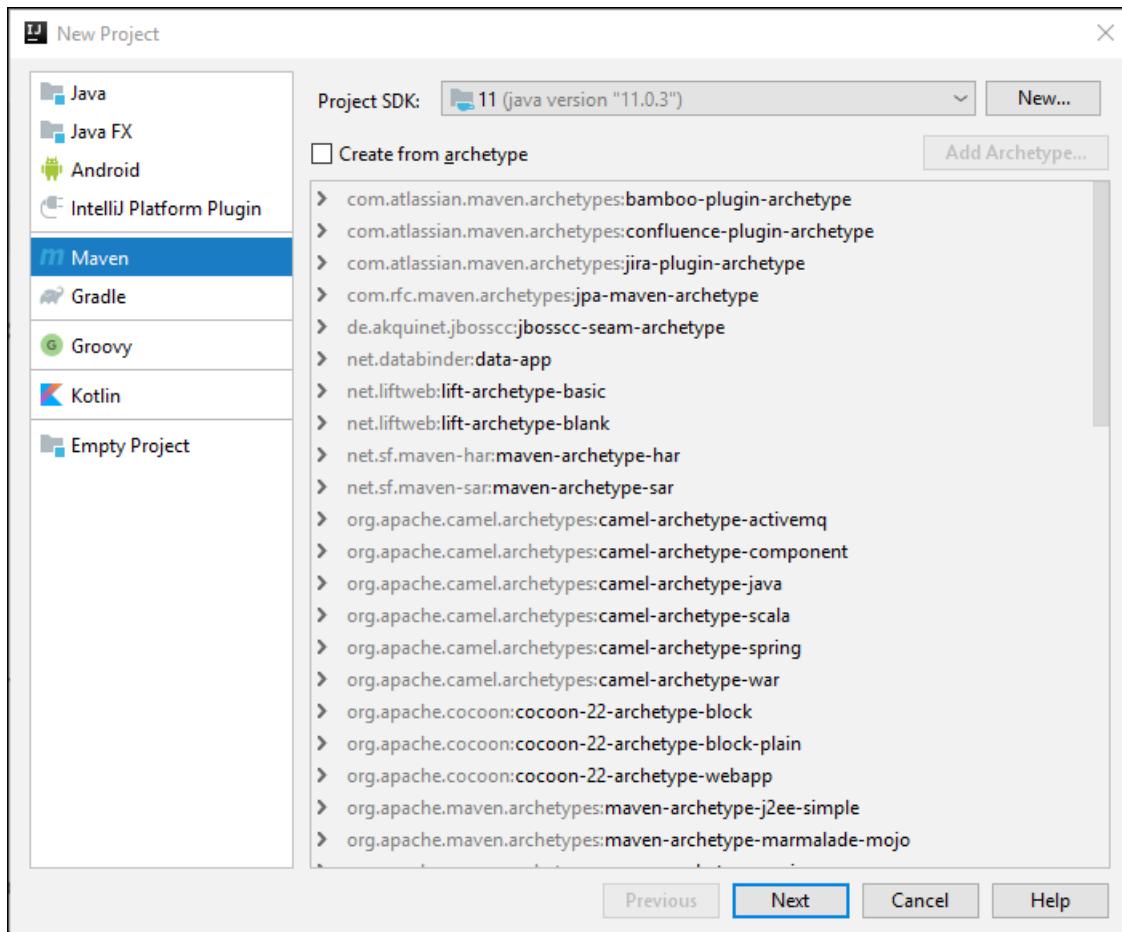
Every request sent to your service requires an API key. Having a valid key establishes trust, on a per request basis, between the application sending the request and the service that handles it.

Set up your environment

Begin by opening IntelliJ IDEA and setting up a new project.

Create the project

1. Open IntelliJ IDEA, and select **Create New Project**.
2. Select **Maven**.
3. In the **Project SDK** list, select the Java 11 SDK.

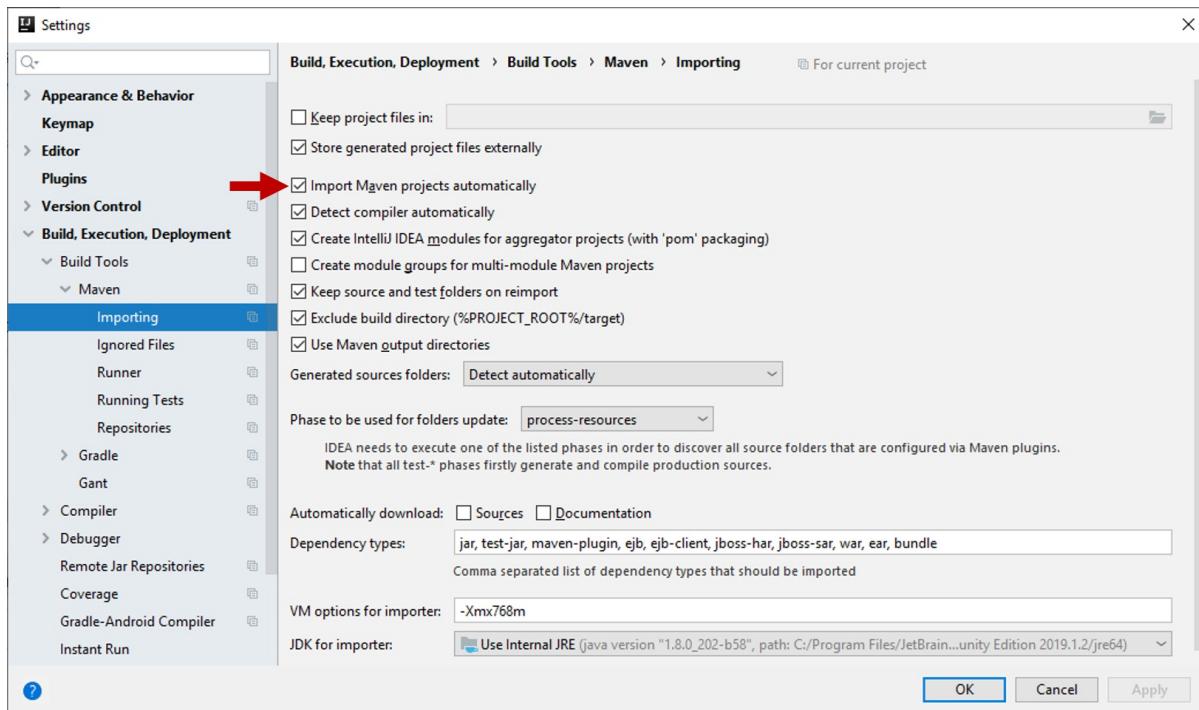


4. For **GroupId** and **ArtifactId**, enter `AzureSearchQuickstart`.

5. Accept the remaining defaults to open the project.

Specify Maven dependencies

1. Select **File > Settings**.
2. In the **Settings** window, select **Build, Execution, Deployment > Build Tools > Maven > Importing**.
3. Select the **Import Maven projects automatically** check box, and click **OK** to close the window. Maven plugins and other dependencies will now be automatically synchronized when you update the pom.xml file in the next step.



4. Open the pom.xml file and replace the contents with the following Maven configuration details. These include references to the [Exec Maven Plugin](#) and a [JSON interface API](#)

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

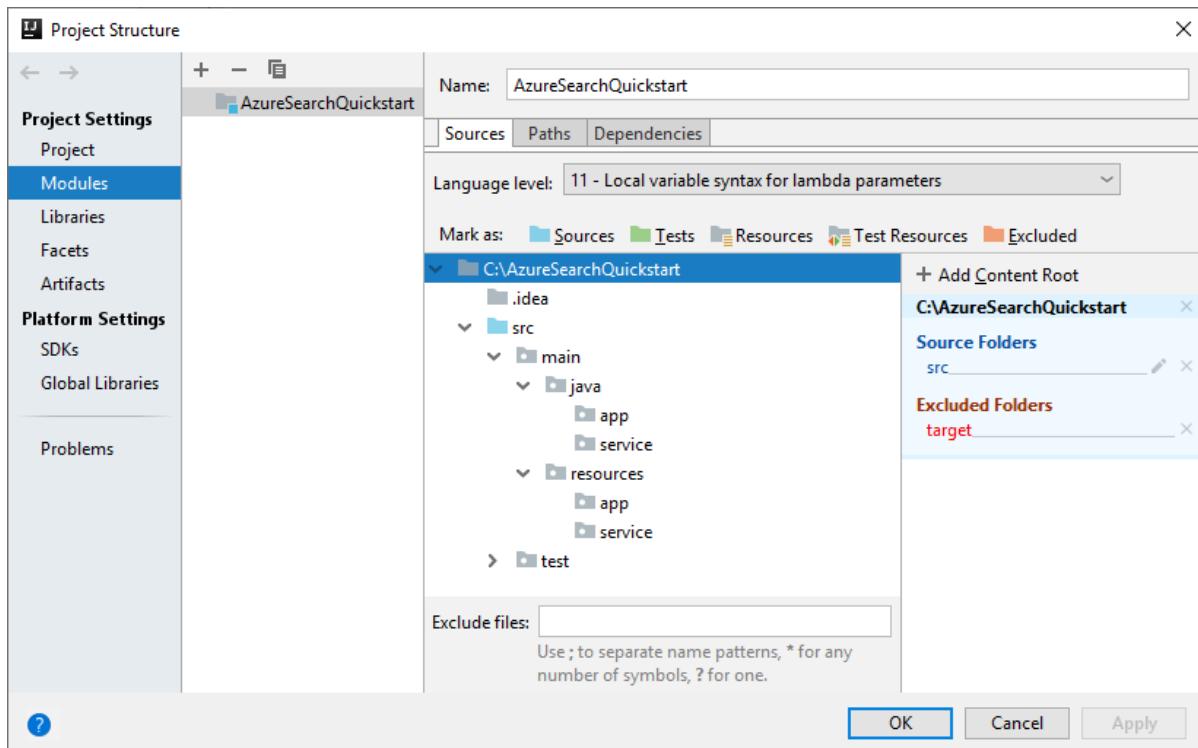
    <groupId>AzureSearchQuickstart</groupId>
    <artifactId>AzureSearchQuickstart</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.6.0</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>exec</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <mainClass>main.java.app.App</mainClass>
                    <cleanupDaemonThreads>false</cleanupDaemonThreads>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.glassfish</groupId>
            <artifactId>javax.json</artifactId>
            <version>1.0.2</version>
        </dependency>
    </dependencies>
</project>

```

Set up the project structure

1. Select **File > Project Structure**.
2. Select **Modules**, and expand the source tree to access the contents of the `src > main` folder.
3. In the `src > main > java` folder, add `app` and `service` folders. To do this, select the `java` folder, press **Alt + Insert**, and then enter the folder name.
4. In the `src > main > resources` folder, add `app` and `service` folders.

When you're done, the project tree should look like the following picture.



- Click **OK** to close the window.

Add Azure Cognitive Search service information

- In the **Project** window, expand the source tree to access the `src > main > resources > app` folder, and add a `config.properties` file. To do this, select the `app` folder, press Alt + Insert, select **File**, and then enter the file name.
- Copy the following settings into the new file and replace `<YOUR-SEARCH-SERVICE-NAME>`, `<YOUR-ADMIN-KEY>`, and `<YOUR-QUERY-KEY>` with your service name and keys. If your service endpoint is `https://mydemo.search.windows.net`, the service name would be `"mydemo"`.

```
SearchServiceName=<YOUR-SEARCH-SERVICE-NAME>
SearchServiceAdminKey=<YOUR-ADMIN-KEY>
SearchServiceQueryKey=<YOUR-QUERY-KEY>
IndexName=hotels-quickstart
ApiVersion=2020-06-30
```

Add the main method

- In the `src > main > java > app` folder, add an `App` class. To do this, select the `app` folder, press Alt + Insert, select **Java Class**, and then enter the class name.
- Open the `App` class and replace the content with the following code. This code contains the `main` method.

The uncommented code reads the search service parameters and uses them to create an instance of the search service client. The search service client code will be added in the next section.

The commented code in this class will be uncommented in a later section of this quickstart.

```
package main.java.app;

import main.java.service.SearchServiceClient;
import java.io.IOException;
import java.util.Properties;

public class App {

    private static Properties loadPropertiesFromResource(String resourcePath) throws IOException {
        Properties properties = new Properties();
        try (InputStream inputStream = getClass().getResourceAsStream(resourcePath)) {
            properties.load(inputStream);
        }
        return properties;
    }

    public static void main(String[] args) {
        Properties properties = loadPropertiesFromResource("config.properties");
        SearchServiceClient client = new SearchServiceClientBuilder()
            .withConnectionStringBuilder(new AzureConnectionStringBuilder(properties))
            .buildClient();
        // ...
    }
}
```

```

private static Properties loadPropertiesFromResource(String resourcePath) throws IOException {
    var inputStream = App.class.getResourceAsStream(resourcePath);
    var configProperties = new Properties();
    configProperties.load(inputStream);
    return configProperties;
}

public static void main(String[] args) {
    try {
        var config = loadPropertiesFromResource("/app/config.properties");
        var client = new SearchServiceClient(
            config.getProperty("SearchServiceName"),
            config.getProperty("SearchServiceAdminKey"),
            config.getProperty("SearchServiceQueryKey"),
            config.getProperty("ApiVersion"),
            config.getProperty("IndexName")
        );
    }

    //Uncomment the next 3 lines in the 1 - Create Index section of the quickstart
    //    if(client.indexExists()){ client.deleteIndex();}
    //    client.createIndex("/service/index.json");
    //    Thread.sleep(1000L); // wait a second to create the index

    //Uncomment the next 2 lines in the 2 - Load Documents section of the quickstart
    //    client.uploadDocuments("/service/hotels.json");
    //    Thread.sleep(2000L); // wait 2 seconds for data to upload

    //Uncomment the following 5 search queries in the 3 - Search an index section of the quickstart
    //    // Query 1
    //    client.logMessage("\n*QUERY
1*****";
    //    client.logMessage("Search for: Atlanta'");
    //    client.logMessage("Return: All fields'");
    //    client.searchPlus("Atlanta");
    //
    //    // Query 2
    //    client.logMessage("\n*QUERY
2*****";
    //    client.logMessage("Search for: Atlanta");
    //    client.logMessage("Return: HotelName, Tags, Address");
    //    SearchServiceClient.SearchOptions options2 = client.createSearchOptions();
    //    options2.select = "HotelName,Tags,Address";
    //    client.searchPlus("Atlanta", options2);
    //
    //    //Query 3
    //    client.logMessage("\n*QUERY
3*****";
    //    client.logMessage("Search for: wifi & restaurant");
    //    client.logMessage("Return: HotelName, Description, Tags");
    //    SearchServiceClient.SearchOptions options3 = client.createSearchOptions();
    //    options3.select = "HotelName,Description,Tags";
    //    client.searchPlus("wifi,restaurant", options3);
    //
    //    // Query 4 -filtered query
    //    client.logMessage("\n*QUERY
4*****";
    //    client.logMessage("Search for: all");
    //    client.logMessage("Filter: Ratings greater than 4");
    //    client.logMessage("Return: HotelName, Rating");
    //    SearchServiceClient.SearchOptions options4 = client.createSearchOptions();
    //    options4.filter="Rating%20gt%204";
    //    options4.select = "HotelName,Rating";
    //    client.searchPlus("",options4);
    //
    //    // Query 5 - top 2 results, ordered by
    //    client.logMessage("\n*QUERY
5*****";
    //    client.logMessage("Search for: boutique");
    //
}

```

```

        //         client.logMessage("Get: top 2 results");
        //         client.logMessage("Order by: Rating in descending order");
        //         client.logMessage("Return: HotelId, HotelName, Category, Rating");
        //         SearchServiceClient.SearchOptions options5 = client.createSearchOptions();
        //         options5.top=2;
        //         options5.orderby = "Rating%20desc";
        //         options5.select = "HotelId,HotelName,Category,Rating";
        //         client.searchPlus("boutique", options5);

    } catch (Exception e) {
        System.err.println("Exception:" + e.getMessage());
        e.printStackTrace();
    }
}
}

```

Add the HTTP operations

1. In the `src` > `main` > `java` > `service` folder, add an `SearchServiceClient` class. To do this, select the `service` folder, press Alt + Insert, select **Java Class**, and then enter the class name.
2. Open the `SearchServiceClient` class, and replace the contents with the following code. This code provides the HTTP operations required to use the Azure Cognitive Search REST API. Additional methods for creating an index, uploading documents, and querying the index will be added in a later section.

```

package main.java.service;

import javax.json.Json;
import javax.net.ssl.HttpsURLConnection;
import java.io.IOException;
import java.io.StringReader;
import java.net.HttpURLConnection;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.charset.StandardCharsets;
import java.util.Formatter;
import java.util.function.Consumer;

/* This class is responsible for implementing HTTP operations for creating the index, uploading
documents and searching the data*/
public class SearchServiceClient {
    private final String _adminKey;
    private final String _queryKey;
    private final String _apiVersion;
    private final String _serviceName;
    private final String _indexName;
    private final static HttpClient client = HttpClient.newHttpClient();

    public SearchServiceClient(String serviceName, String adminKey, String queryKey, String apiVersion,
String indexName) {
        this._serviceName = serviceName;
        this._adminKey = adminKey;
        this._queryKey = queryKey;
        this._apiVersion = apiVersion;
        this._indexName = indexName;
    }

    private static HttpResponse<String> sendRequest(HttpRequest request) throws IOException,
InterruptedException {
        logMessage(String.format("%s: %s", request.method(), request.uri()));
        return client.send(request, HttpResponse.BodyHandlers.ofString());
    }

    private static URI buildURI(Consumer<Formatter> fmtFn)
    {

```

```

        Formatter strFormatter = new Formatter();
        fmtFn.accept(strFormatter);
        String url = strFormatter.out().toString();
        strFormatter.close();
        return URI.create(url);
    }

    public static void logMessage(String message) {
        System.out.println(message);
    }

    public static boolean isSuccessResponse(HttpResponse<String> response) {
        try {
            int responseCode = response.statusCode();

            logMessage("\n Response code = " + responseCode);

            if (responseCode == HttpURLConnection.HTTP_OK || responseCode ==
HttpURLConnection.HTTP_ACCEPTED
                || responseCode == HttpURLConnection.HTTP_NO_CONTENT || responseCode ==
HttpsURLConnection.HTTP_CREATED) {
                return true;
            }

            // We got an error
            var msg = response.body();
            if (msg != null) {
                logMessage(String.format("\n MESSAGE: %s", msg));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return false;
    }

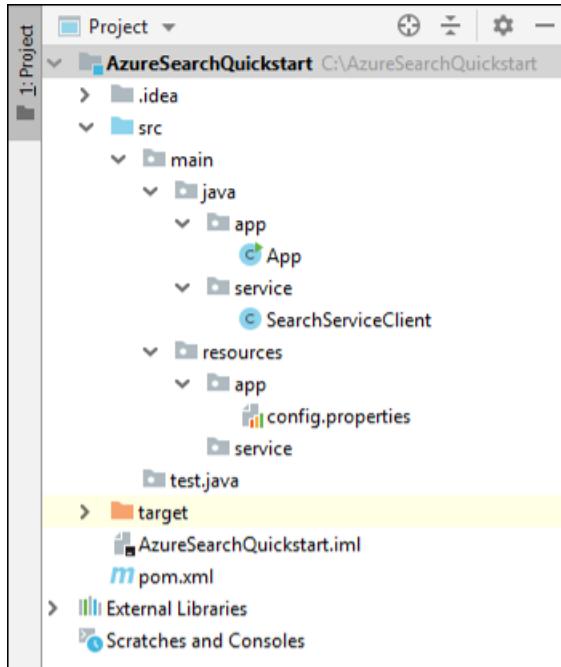
    public static HttpRequest httpRequest(URI uri, String key, String method, String contents) {
        contents = contents == null ? "" : contents;
        var builder = HttpRequest.newBuilder();
        builder.uri(uri);
        builder.setHeader("content-type", "application/json");
        builder.setHeader("api-key", key);

        switch (method) {
            case "GET":
                builder = builder.GET();
                break;
            case "HEAD":
                builder = builder.GET();
                break;
            case "DELETE":
                builder = builder.DELETE();
                break;
            case "PUT":
                builder = builder.PUT(HttpRequest.BodyPublishers.ofString(contents));
                break;
            case "POST":
                builder = builder.POST(HttpRequest.BodyPublishers.ofString(contents));
                break;
            default:
                throw new IllegalArgumentException(String.format("Can't create request for method '%s'", method));
        }
        return builder.build();
    }
}

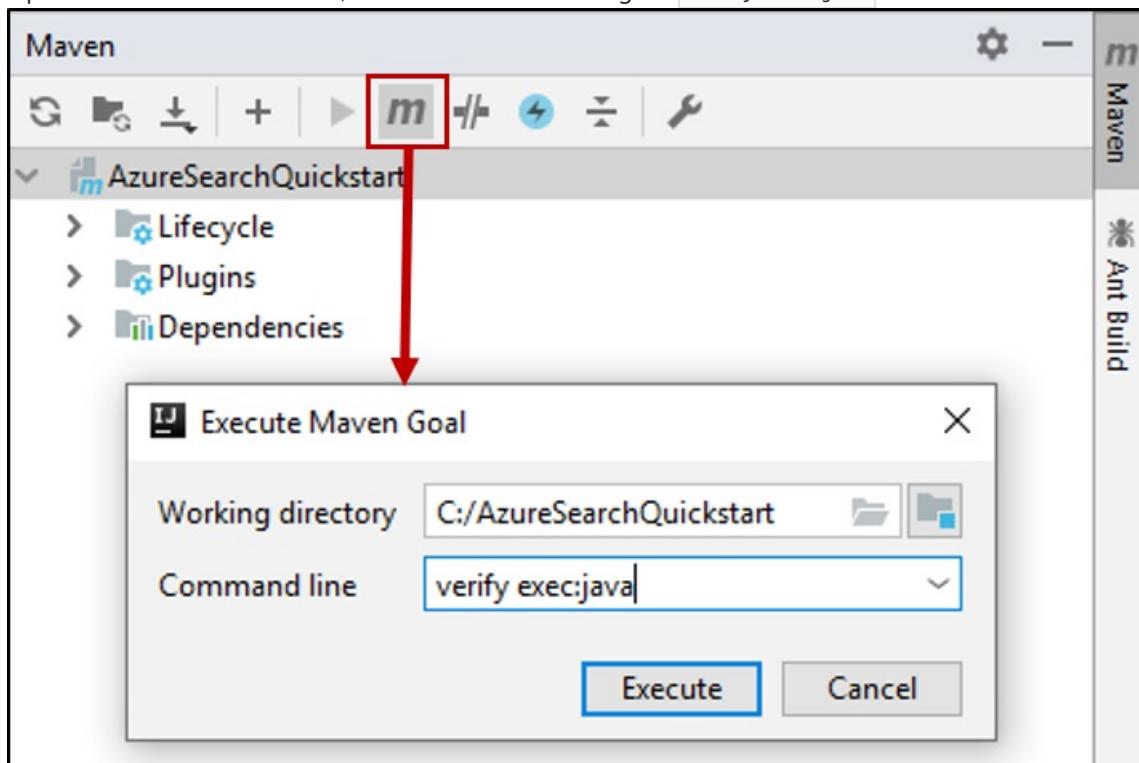
```

Build the project

1. Verify that your project has the following structure.



2. Open the **Maven** tool window, and execute this maven goal: `verify exec:java`



When processing completes, look for a BUILD SUCCESS message followed by a zero (0) exit code.

1 - Create index

The hotels index definition contains simple fields and one complex field. Examples of a simple field are "HotelName" or "Description". The "Address" field is a complex field because it has subfields, such as "Street Address" and "City". In this quickstart, the index definition is specified using JSON.

1. In the **Project** window, expand the source tree to access the `src > main > resources > service` folder, and add an `index.json` file. To do this, select the `app` folder, press Alt + Insert, select **File**, and then enter

the file name.

2. Open the `index.json` file and insert the following index definition.

```
{  
  "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": "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": "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
    }
  ]
}
]
}

```

The index name will be "hotels-quickstart". Attributes on the index fields determine how the indexed data can be searched in an application. For example, the `Issearchable` attribute must be assigned to every field that should be included in a full text search. To learn more about attributes, see [Fields collection and field attributes](#).

The `Description` field in this index uses the optional `analyzer` property to override the default Lucene language analyzer. The `Description_fr` field is using the French Lucene analyzer `fr.lucene` because it stores French text. The `Description` is using the optional Microsoft language analyzer `en.lucene`. To learn

more about analyzers, see [Analyzers for text processing in Azure Cognitive Search](#).

3. Add the following code to the `SearchServiceClient` class. These methods build Azure Cognitive Search REST service URLs that create and delete an index, and that determine if an index exists. The methods also make the HTTP request.

```
public boolean indexExists() throws IOException, InterruptedException {
    logMessage("\n Checking if index exists...");
    var uri = buildURI(strFormatter -> strFormatter.format(
        "https://$s.search.windows.net/indexes/$s/docs?api-version=%s&search=*",
        _serviceName,_indexName,_apiVersion));
    var request = httpRequest(uri, _adminKey, "HEAD", "");
    var response = sendRequest(request);
    return isSuccessResponse(response);
}

public boolean deleteIndex() throws IOException, InterruptedException {
    logMessage("\n Deleting index...");
    var uri = buildURI(strFormatter -> strFormatter.format(
        "https://$s.search.windows.net/indexes/$s?api-version=%s",
        _serviceName,_indexName,_apiVersion));
    var request = httpRequest(uri, _adminKey, "DELETE", "*");
    var response = sendRequest(request);
    return isSuccessResponse(response);
}

public boolean createIndex(String indexDefinitionFile) throws IOException, InterruptedException {
    logMessage("\n Creating index...");
    //Build the search service URL
    var uri = buildURI(strFormatter -> strFormatter.format(
        "https://$s.search.windows.net/indexes/$s?api-version=%s",
        _serviceName,_indexName,_apiVersion));
    //Read in index definition file
    var inputStream = SearchServiceClient.class.getResourceAsStream(indexDefinitionFile);
    var indexDef = new String(inputStream.readAllBytes(), StandardCharsets.UTF_8);
    //Send HTTP PUT request to create the index in the search service
    var request = httpRequest(uri, _adminKey, "PUT", indexDef);
    var response = sendRequest(request);
    return isSuccessResponse(response);
}
```

4. Uncomment the following code in the `App` class. This code deletes the "hotels-quickstart" index, if it exists, and creates a new index based on the index definition in the "index.json" file.

A one-second pause is inserted after the index creation request. This pause ensures that the index is created before you upload documents.

```
if (client.indexExists()) { client.deleteIndex();}
client.createIndex("/service/index.json");
Thread.sleep(1000L); // wait a second to create the index
```

5. Open the **Maven** tool window, and execute this maven goal: `verify exec:java`

As the code runs, look for a "Creating index" message followed by a 201 response code. This response code confirms that the index was created. The run should end with a BUILD SUCCESS message and a zero (0) exit code.

2 - Load documents

1. In the **Project** window, expand the source tree to access the `src` > `main` > `resources` > `service` folder,

and add an `hotels.json` file. To do this, select the `app` folder, press Alt + Insert, select File, and then enter the file name.

2. Insert the following hotel documents into the file.

```
{
  "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.",
      "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": 3.60
    }
  ]
}
```

```

    "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"
    }
}
]
}

```

3. Insert the following code into the `SearchServiceClient` class. This code builds the REST service URL to upload the hotel documents to the index, and then makes the HTTP POST request.

```

public boolean uploadDocuments(String documentsFile) throws IOException, InterruptedException {
    logMessage("\n Uploading documents...");
    //Build the search service URL
    var endpoint = buildURI(strFormatter -> strFormatter.format(
        "https://%s.search.windows.net/indexes/%s/docs/index?api-version=%s",
        _serviceName,_indexName,_apiVersion));
    //Read in the data to index
    var inputStream = SearchServiceClient.class.getResourceAsStream(documentsFile);
    var documents = new String(inputStream.readAllBytes(), StandardCharsets.UTF_8);
    //Send HTTP POST request to upload and index the data
    var request = httpRequest(endpoint, _adminKey, "POST", documents);
    var response = sendRequest(request);
    return isSuccessResponse(response);
}

```

4. Uncomment the following code in the `App` class. This code uploads the documents in "hotels.json" to the index.

```

client.uploadDocuments("/service/hotels.json");
Thread.sleep(2000L); // wait 2 seconds for data to upload

```

A two-second pause is inserted after the upload request to ensure that the document loading process completes before you query the index.

5. Open the **Maven** tool window, and execute this maven goal: `verify exec:java`

Because you created a "hotels-quickstart" index in the previous step, the code will now delete it and recreate it again before loading the hotel documents.

As the code runs, look for an "Uploading documents" message followed by a 200 response code. This response code confirms that the documents were uploaded to the index. The run should end with a BUILD SUCCESS message and a zero (0) exit code.

3 - Search an index

Now that you've loaded the hotels documents, you can create search queries to access the hotels data.

1. Add the following code to the `SearchServiceClient` class. This code builds Azure Cognitive Search REST service URLs to search the indexed data and prints the search results.

The `SearchOptions` class and `createSearchOptions` method let you specify a subset of the available Azure Cognitive Search REST API query options. For more information on the REST API query options, see [Search Documents \(Azure Cognitive Search REST API\)](#).

The `SearchPlus` method creates the search query URL, makes the search request, and then prints the results to the console.

```

public SearchOptions createSearchOptions() { return new SearchOptions();}

//Defines available search parameters that can be set
public static class SearchOptions {

    public String select = "";
    public String filter = "";
    public int top = 0;
    public String orderby= "";
}

//Concatenates search parameters to append to the search request
private String createOptionsString(SearchOptions options)
{
    String optionsString = "";
    if (options != null) {
        if (options.select != "")
            optionsString = optionsString + "&$select=" + options.select;
        if (options.filter != "")
            optionsString = optionsString + "&$filter=" + options.filter;
        if (options.top != 0)
            optionsString = optionsString + "&$top=" + options.top;
        if (options.orderby != "")
            optionsString = optionsString + "&$orderby=" + options.orderby;
    }
    return optionsString;
}

public void searchPlus(String queryString)
{
    searchPlus( queryString, null);
}

public void searchPlus(String queryString, SearchOptions options) {

    try {
        String optionsString = createOptionsString(options);
        var uri = buildURI(strFormatter -> strFormatter.format(
            "https://%s.search.windows.net/indexes/%s/docs?api-version=%s&search=%s",
            _serviceName, _indexName, _apiVersion, queryString, optionsString));
        var request = httpRequest(uri, _queryKey, "GET", null);
        var response = sendRequest(request);
        var jsonReader = Json.createReader(new StringReader(response.body()));
        var jsonArray = jsonReader.readObject().getJSONArray("value");
        var resultsCount = jsonArray.size();
        logMessage("Results:\nCount: " + resultsCount);
        for (int i = 0; i <= resultsCount - 1; i++) {
            logMessage(jsonArray.get(i).toString());
        }

        jsonReader.close();

    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

2. In the `App` class, uncomment the following code. This code sets up five different queries, including the search text, query parameters, and data fields to return.

```

// Query 1
client.logMessage("\n*QUERY 1*****");
client.logMessage("Search for: Atlanta");
client.logMessage("Return: All fields'");
client.searchPlus("Atlanta");

// Query 2
client.logMessage("\n*QUERY 2*****");
client.logMessage("Search for: Atlanta");
client.logMessage("Return: HotelName, Tags, Address");
SearchServiceClient.SearchOptions options2 = client.createSearchOptions();
options2.select = "HotelName,Tags,Address";
client.searchPlus("Atlanta", options2);

//Query 3
client.logMessage("\n*QUERY 3*****");
client.logMessage("Search for: wifi & restaurant");
client.logMessage("Return: HotelName, Description, Tags");
SearchServiceClient.SearchOptions options3 = client.createSearchOptions();
options3.select = "HotelName,Description,Tags";
client.searchPlus("wifi,restaurant", options3);

// Query 4 -filtered query
client.logMessage("\n*QUERY 4*****");
client.logMessage("Search for: all");
client.logMessage("Filter: Ratings greater than 4");
client.logMessage("Return: HotelName, Rating");
SearchServiceClient.SearchOptions options4 = client.createSearchOptions();
options4.filter="Rating%20gt%204";
options4.select = "HotelName,Rating";
client.searchPlus("*",options4);

// Query 5 - top 2 results, ordered by
client.logMessage("\n*QUERY 5*****");
client.logMessage("Search for: boutique");
client.logMessage("Get: Top 2 results");
client.logMessage("Order by: Rating in descending order");
client.logMessage("Return: HotelId, HotelName, Category, Rating");
SearchServiceClient.SearchOptions options5 = client.createSearchOptions();
options5.top=2;
options5.orderby = "Rating%20desc";
options5.select = "HotelId,HotelName,Category,Rating";
client.searchPlus("boutique", options5);

```

There are two [ways of matching terms in a query](#): full-text search, and filters. A full-text search query searches for one or more terms in `IsSearchable` fields in your index. A filter is a boolean expression that is evaluated over `IsFilterable` fields in an index. You can use full-text search and filters together or separately.

3. Open the **Maven** tool window, and execute this maven goal: `verify exec:java`

Look for a summary of each query and its results. The run should complete with BUILD SUCCESS message and a zero (0) exit code.

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.

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 Java quickstart, you worked through a series of tasks to create an index, load it with documents, and run queries. If you are comfortable with the basic concepts, we recommend the following article that lists indexer operations in REST.

[Indexer operations](#)

Quickstart: Create an Azure Cognitive Search index in Node.js using REST APIs

10/4/2020 • 19 minutes to read • [Edit Online](#)

Create a Node.js application that creates, loads, and queries an Azure Cognitive 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

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

- [Node.js](#)
- [NPM](#) should be installed by Nodejs
- A sample index structure and matching documents are provided in this article, or from the [quickstart directory of the repo](#)
- [Create an Azure Cognitive Search service](#) or [find an existing service](#) under your current subscription. You can use a free service for this quickstart.

Recommended:

- [Visual Studio Code](#)
- [Prettier](#) and [ESLint](#) extensions for VSCode.

Get keys and URLs

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 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.

Get the query key as well. It's a best practice to issue query requests with read-only access.

The screenshot shows the Microsoft Azure portal interface. The left sidebar has a tree view with 'mydemo' selected. Under 'mydemo', 'Keys' is highlighted. The main content area shows the 'Keys' section for the search service. It includes fields for 'Primary admin key' (containing the placeholder) and 'Secondary admin key'. Below that is a table titled 'Manage query keys' with a single row. The row has columns for 'NAME' and 'KEY', both containing the placeholder. There are buttons for 'Add' and a three-dot menu.

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.

Set up your environment

Begin by opening a Powershell console or other environment in which you've installed Node.js.

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

```
mkdir quickstart
cd quickstart
```

2. Initialize an empty project with NPM by running `npm init`. Accept the default values, except for the License, which you should set to "MIT".

3. Add packages that will be depended on by the code and aid in development:

```
npm install nconf node-fetch
npm install --save-dev eslint eslint-config-prettier eslint-config-airbnb-base eslint-plugin-import
prettier
```

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

```
{
  "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",
    "Azure_Search"
  ],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "nconf": "^0.10.0",
    "node-fetch": "^2.6.0"
  },
  "devDependencies": {
    "eslint": "^6.1.0",
    "eslint-config-airbnb-base": "^13.2.0",
    "eslint-config-prettier": "^6.0.0",
    "eslint-plugin-import": "^2.18.2",
    "prettier": "^1.18.2"
  }
}
```

5. Create a file `azure_search_config.json` to hold your search service data:

```
{
  "serviceName" : "[SEARCH_SERVICE_NAME]",
  "adminKey" : "[ADMIN_KEY]",
  "queryKey" : "[QUERY_KEY]",
  "indexName" : "hotels-quickstart"
}
```

Replace the `[SERVICE_NAME]` value with the name of your search service. Replace `[ADMIN_KEY]` and `[QUERY_KEY]` with the key values you recorded earlier.

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](#).

Add the following to `hotels_quickstart_index.json` or [download the file](#).

```
{
  "name": "hotels-quickstart",
  "fields": [
    {
      "name": "HotelId",
      "type": "Edm.String",
      "key": true,
      "filterable": true
    },
    {
      "name": "HotelName",
      "type": "Edm.String",
      "key": false,
      "filterable": true
    }
  ]
}
```

```
        "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": "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"
    ]
}
]
}

```

It's good practice to separate the specifics of a particular scenario from code that will be broadly applicable. The `AzureSearchClient` class defined in the file `AzureSearchClient.js` will know how to construct request URLs, make a request using the Fetch API, and react to the status code of the response.

Begin working on `AzureSearchClient.js` by importing the `node-fetch` package and creating a simple class. Isolate the changeable parts of the `AzureSearchClient` class by passing to its constructor the various configuration values:

```

const fetch = require('node-fetch');

class AzureSearchClient {
    constructor(searchServiceName, adminKey, queryKey, indexName) {
        this.searchServiceName = searchServiceName;
        this.adminKey = adminKey;
        // The query key is used for read-only requests and so can be distributed with less risk of abuse.
        this.queryKey = queryKey;
        this.indexName = indexName;
        this.apiVersion = '2020-06-30';
    }

    // All methods go inside class body here!
}

module.exports = AzureSearchClient;

```

The first responsibility of the class is to know how to construct URLs to which to send the various requests. Build these URLs with instance methods that use the configuration data passed to the class constructor. Notice that the URL they construct is specific to an API version and must have an argument specifying that version (in this application, `2020-06-30`).

The first of these methods will return the URL for the index itself. Add the following method inside the class body:

```

getIndexUrl() { return `https://${this.searchServiceName}.search.windows.net/indexes/${this.indexName}?api-
version=${this.apiVersion}`; }

```

The next responsibility of `AzureSearchClient` is making an asynchronous request with the Fetch API. The asynchronous static method `request` takes a URL, a string specifying the HTTP method ("GET", "PUT", "POST", "DELETE"), the key to be used in the request, and an optional JSON object. The `headers` variable maps the `queryKey` (whether the admin key or the read-only query key) to the "api-key" HTTP request header. The request options always contain the `method` to be used and the `headers`. If `bodyJson` isn't `null`, the body of the HTTP request is set to the string representation of `bodyJson`. The `request` method returns the Fetch API's Promise to execute the HTTP request.

```

static async request(url, method, apiKey, bodyJson = null) {
    // Uncomment the following for request details:
    /*
    console.log(`\n${method} ${url}`);
    console.log(`\nKey ${apiKey}`);
    if (bodyJson !== null) {
        console.log(`\ncontent: ${JSON.stringify(bodyJson, null, 4)})`);
    }
    */

    const headers = {
        'content-type' : 'application/json',
        'api-key' : apiKey
    };
    const init = bodyJson === null ?
    {
        method,
        headers
    } :
    {
        method,
        headers,
        body : JSON.stringify(bodyJson)
    };
    return fetch(url, init);
}

```

For demo purposes, just throw an exception if the HTTP request is not a success. In a real application, you would probably do some logging and diagnosis of the HTTP status code in the `response` from the search service request.

```

static throwOnHttpError(response) {
    const statusCode = response.status;
    if (statusCode >= 300){
        console.log(`Request failed: ${JSON.stringify(response, null, 4)})`);
        throw new Error(`Failure in request. HTTP Status was ${statusCode}`);
    }
}

```

Finally, add the methods to detect, delete, and create the Azure Cognitive Search index. These methods all have the same structure:

- Get the endpoint to which the request will be made.
- Generate the request with the appropriate endpoint, HTTP verb, API key, and, if appropriate, a JSON body.
`indexExistsAsync()` and `deleteIndexAsync()` do not have a JSON body, but `createIndexAsync(definition)` does.
- `await` the response to the request.
- Act on the status code of the response.
- Return a Promise of some appropriate value (a Boolean, `this`, or the query results).

```

async indexExistsAsync() {
    console.log("\n Checking if index exists...");
    const endpoint = this.getIndexUrl();
    const response = await AzureSearchClient.request(endpoint, "GET", this.adminKey);
    // Success has a few likely status codes: 200 or 204 (No Content), but accept all in 200 range...
    const exists = response.status >= 200 && response.status < 300;
    return exists;
}

async deleteIndexAsync() {
    console.log("\n Deleting existing index...");
    const endpoint = this.getIndexUrl();
    const response = await AzureSearchClient.request(endpoint, "DELETE", this.adminKey);
    AzureSearchClient.throwOnHttpError(response);
    return this;
}

async createIndexAsync(definition) {
    console.log("\n Creating index...");
    const endpoint = this.getIndexUrl();
    const response = await AzureSearchClient.request(endpoint, "PUT", this.adminKey, definition);
    AzureSearchClient.throwOnHttpError(response);
    return this;
}

```

Confirm that your methods are inside the class and that you're exporting the class. The outermost scope of `AzureSearchClient.js` should be:

```

const fetch = require('node-fetch');

class AzureSearchClient {
    // ... code here ...
}

module.exports = AzureSearchClient;

```

An object-oriented class was a good choice for the potentially reusable `AzureSearchClient.js` module, but isn't necessary for the main program, which you should put in a file called `index.js`.

Create `index.js` and begin by bringing in:

- The `nconf` package, which gives you flexibility for specifying the configuration with JSON, environment variables, or command-line arguments.
- The data from the `hotels_quickstart_index.json` file.
- The `AzureSearchClient` module.

```

const nconf = require('nconf');

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

```

The [nconf package](#) allows you to specify configuration data in a variety of formats, such as environment variables or the command line. This sample uses `nconf` in a basic manner to read the file `azure_search_config.json` and return that file's contents as a dictionary. Using `nconf`'s `get(key)` function, you can do a quick check that the configuration information has been properly customized. Finally, the function returns the configuration:

```

function getAzureConfiguration() {
    const config = nconf.file({ file: 'azure_search_config.json' });
    if (config.get('serviceName') === '[SEARCH_SERVICE_NAME]' ) {
        throw new Error("You have not set the values in your azure_search_config.json file. Change them to match your search service's values.");
    }
    return config;
}

```

The `sleep` function creates a `Promise` that resolves after a specified amount of time. Using this function allows the app to pause while waiting for asynchronous index operations to complete and become available. Adding such a delay is typically only necessary in demos, tests, and sample applications.

```

function sleep(ms) {
    return(
        new Promise(function(resolve, reject) {
            setTimeout(function() { resolve(); }, ms);
        })
    );
}

```

Finally, specify and call the main asynchronous `run` function. This function calls the other functions in order, awaiting as necessary to resolve `Promise`s.

- Retrieve the configuration with the `getAzureConfiguration()` you wrote previously
- Create a new `AzureSearchClient` instance, passing in values from your configuration
- Check if the index exists and, if it does, delete it
- Create an index using the `indexDefinition` loaded from `hotels_quickstart_index.json`

```

const run = async () => {
    try {
        const cfg = getAzureConfiguration();
        const client = new AzureSearchClient(cfg.get("serviceName"), cfg.get("adminKey"), cfg.get("queryKey"),
        cfg.get("indexName"));

        const exists = await client.indexExistsAsync();
        await exists ? client.deleteIndexAsync() : Promise.resolve();
        // Deleting index can take a few seconds
        await sleep(2000);
        await client.createIndexAsync(indexDefinition);
    } catch (x) {
        console.log(x);
    }
}

run();

```

Don't forget that final call to `run()`! It's the entrance point to your program when you run `node index.js` in the next step.

Notice that `AzureSearchClient.indexExistsAsync()` and `AzureSearchClient.deleteIndexAsync()` do not take parameters. These functions call `AzureSearchClient.request()` with no `bodyJson` argument. Within `AzureSearchClient.request()`, since `bodyJson == null` is `true`, the `init` structure is set to be just the HTTP verb ("GET" for `indexExistsAsync()` and "DELETE" for `deleteIndexAsync()`) and the headers, which specify the request key.

In contrast, the `AzureSearchClient.createIndexAsync(indexDefinition)` method *does* take a parameter. The `run` function in `index.js`, passes the contents of the file `hotels_quickstart_index.json` to the

`AzureSearchClient.createIndexAsync(indexDefinition)` method. The `createIndexAsync()` method passes this definition to `AzureSearchClient.request()`. In `AzureSearchClient.request()`, since `bodyJson === null` is now `false`, the `init` structure includes not only the HTTP verb ("PUT") and the headers, but sets the `body` to the index definition data.

Prepare and run the sample

Use a terminal window for the following commands.

1. Navigate to the folder that contains the `package.json` file and the rest of your code.
2. Install the packages for the sample with `npm install`. This command will download the packages upon which the code depends.
3. Run your program with `node index.js`.

You should see a series of messages describing the actions being taken by the program. If you want to see more detail of the requests, you can uncomment the [lines at the beginning of the `AzureSearchClient.request()` method]https://github.com/Azure-Samples/azure-search-javascript-samples/blob/master/quickstart/AzureSearchClient.js#L21-L27) in `AzureSearchClient.js`.

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

The screenshot shows the Azure portal interface for a search service. On the left, there's a sidebar with various navigation options: Overview (highlighted with a red box), Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick start, Keys, Scale, Search traffic analytics, Identity, Properties, Locks, and Export template. The main content area has a header with a 'Get 99.9% availability guaranteed with 3 replicas or more.' message. It displays resource group information (Resource group: [change]), status (Running), location (West US 2), subscription information (Subscription: [change], Subscription ID: [REDACTED]), and pricing tier (Standard). Below this is a 'Tags' section with a 'Click here to add tags' button. At the bottom of the main content area, there are tabs: Usage, Monitoring, Indexes (highlighted with a red box), Indexers, Data sources, and Skillsets. Under the Indexes tab, there's a table with columns: NAME, DOCUMENT COUNT, and STORAGE SIZE. One row is shown: hotels-quickstart, 0, 0 B.

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 need to POST such data to the index. This uses a different endpoint than the operations done in the previous step. Open `AzureSearchClient.js` and add the following method after `getIndexUrl()`:

```
getPostDataUrl() { return  
`https://${this.searchServiceName}.search.windows.net/indexes/${this.indexName}/docs/index?api-  
version=${this.apiVersion}`; }
```

Like `AzureSearchClient.createIndexAsync(definition)`, you need a function that calls `AzureSearchClient.request()` and passes in the hotel data to be its body. In `AzureSearchClient.js` add `postDataAsync(hotelsData)` after `createIndexAsync(definition)`:

```

async postDataAsync(hotelsData) {
    console.log("\n Adding hotel data...");
    const endpoint = this.getPostDataUrl();
    const response = await AzureSearchClient.request(endpoint, "POST", this.adminKey, hotelsData);
    AzureSearchClient.throwOnHttpError(response);
    return this;
}

```

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:

```
{
    "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"
    }
}
]
}

```

To load this data into your program, modify `index.js` by adding the line referring to `hotelData` near the top:

```

const nconf = require('nconf');

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

```

Now modify the `run()` function in `index.js`. It can take a few seconds for the index to become available, so add a 2-second pause before calling `AzureSearchClient.postDataAsync(hotelData)`:

```

const run = async () => {
    try {
        const cfg = getAzureConfiguration();
        const client = new AzureSearchClient(cfg.get("serviceName"), cfg.get("adminKey"), cfg.get("queryKey"),
        cfg.get("indexName"));

        const exists = await client.indexExistsAsync();
        await exists ? client.deleteIndexAsync() : Promise.resolve();
        // Deleting index can take a few seconds
        await sleep(2000);
        await client.createIndexAsync(indexDefinition);
        // Index availability can take a few seconds
        await sleep(2000);
        await client.postDataAsync(hotelData);
    } catch (x) {
        console.log(x);
    }
}

```

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 message about deleting it before the app creates the new index and posts data to it.

3 - Search an index

Return to the **Indexes** tab in the **Overview** of your search service on the Azure portal. Your index now contains four documents and consumes some amount of storage (it may take a few minutes for the UI to properly reflect the underlying state of the index). Click on the index name to be taken to the **Search Explorer**. This page allows you to experiment with data queries. Try searching on a query string of `*&$count=true` and you should get back all your documents and the number of results. Try with the query string

`historic&highlight=Description&$filter=Rating gt 4` and you should get back a single document, with the word "historic" wrapped in `` tags. Read more about [how to compose a query in Azure Cognitive Search](#).

Reproduce these queries in code by opening `index.js` and adding this code near the top:

```

const queries = [
    "*&$count=true",
    "historic&highlight=Description&$filter=Rating gt 4&"
];

```

In the same `index.js` file, write the `doQueriesAsync()` function shown below. This function takes an `AzureSearchClient` object and applies the `AzureSearchClient.queryAsync` method to each of the values in the `queries` array. It uses the `Promise.all()` function to return a single `Promise` that only resolves when all of the queries have resolved. The call to `JSON.stringify(body, null, 4)` formats the query result to be more readable.

```

async function doQueriesAsync(client) {
    return Promise.all(
        queries.map( async query => {
            const result = await client.queryAsync(query);
            const body = await result.json();
            const str = JSON.stringify( body, null, 4);
            console.log(`Query: ${query} \n ${str}`);
        })
    );
}

```

Modify the `run()` function to pause long enough for the indexer to work and then to call the

```
doQueriesAsync(client) function:
```

```
const run = async () => {
  try {
    const cfg = getAzureConfiguration();
    const client = new AzureSearchClient(cfg.get("serviceName"), cfg.get("adminKey"), cfg.get("queryKey"),
      cfg.get("indexName"));

    const exists = await client.indexExistsAsync();
    await exists ? client.deleteIndexAsync() : Promise.resolve();
    // Deleting index can take a few seconds
    await sleep(2000);
    await client.createIndexAsync(indexDefinition);
    // Index availability can take a few seconds
    await sleep(2000);
    await client.postDataAsync(hotelData);
    // Data availability can take a few seconds
    await sleep(5000);
    await doQueriesAsync(client);
  } catch (x) {
    console.log(x);
  }
}
```

To implement `AzureSearchClient.queryAsync(query)`, edit the file `AzureSearchClient.js`. Searching requires a different endpoint, and the search terms become URL arguments, so add the function `getSearchUrl(searchTerm)` alongside the `getIndexUrl()` and `getPostDataUrl()` methods you've already written.

```
getSearchUrl(searchTerm) { return
`https://${this.searchServiceName}.search.windows.net/indexes/${this.indexName}/docs?api-
version=${this.apiVersion}&search=${searchTerm}&searchMode=all` ; }
```

The `queryAsync(searchTerm)` function also goes in `AzureSearchClient.js` and follows the same structure as `postDataAsync(data)` and the other querying functions:

```
async queryAsync(searchTerm) {
  console.log("\n Querying...")
  const endpoint = this.getSearchUrl(searchTerm);
  const response = await AzureSearchClient.request(endpoint, "GET", this.queryKey);
  AzureSearchClient.throwOnHttpError(response);
  return response;
}
```

Search is done with the "GET" verb and no body, since the search term is part of the URL. Notice that `queryAsync(searchTerm)` uses `this.queryKey`, unlike the other functions that used the admin key. Query keys, as the name implies, can only be used for querying the index and can't be used to modify the index in any way. Query keys are therefore safer to distribute to client applications.

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.

About the sample

The sample uses a small amount of hotel data, sufficient to demonstrate the basics of creating and querying an Azure Cognitive Search index.

The `AzureSearchClient` class encapsulates the configuration, URLs, and basic HTTP requests for the search service. The `index.js` file loads the configuration data for the Azure Cognitive Search service, the hotel data that will be uploaded for indexing, and, in its `run` function, orders, and executes the various operations.

The overall behavior of the `run` function is to delete the Azure Cognitive Search index if it exists, create the index, add some data, and perform some queries.

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 Node.js quickstart, you worked through a series of tasks to create an index, load it with documents, and run queries. We did certain steps, such as reading the configuration and defining the queries, in the simplest possible way. In a real application, you would want to put those concerns in separate modules that would provide flexibility and encapsulation.

If you already have some background in Azure Cognitive Search, you can use this sample as a springboard for trying suggesters (type-ahead or autocomplete queries), filters, and faceted navigation. If you're new to Azure Cognitive Search, we recommend trying other tutorials to develop an understanding of what you can create. Visit our [documentation page](#) to find more resources.

[Call Azure Cognitive Search from a WebPage using Javascript](#)

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

10/4/2020 • 9 minutes to read • [Edit Online](#)

This article explains how to formulate REST API requests interactively using the [Azure Cognitive Search REST APIs](#) and an API client for sending and receiving requests. With an API client and these instructions, you can send requests and view responses before writing any code.

The article uses the Postman application. 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

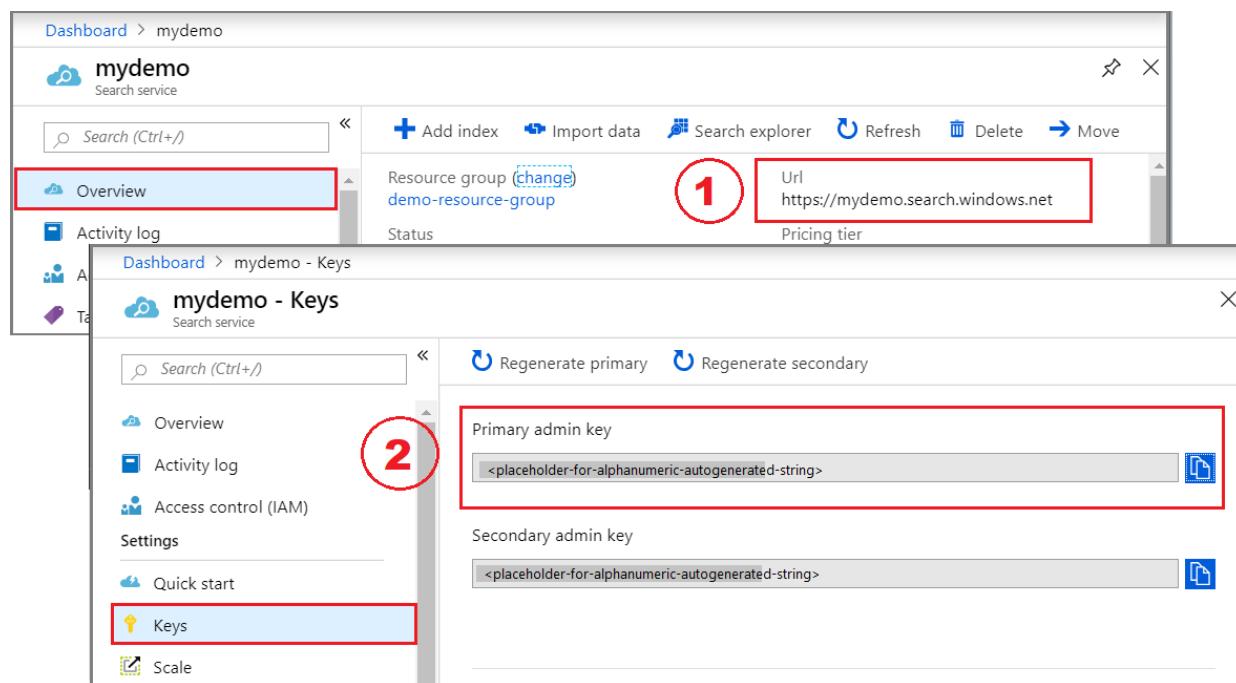
The following services and tools are required for this quickstart.

- [Postman desktop app](#) is 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.

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.

Connect to Azure Cognitive Search

In this section, use your web tool of choice to set up connections to Azure Cognitive Search. Each tool persists request header information for the session, which means you only have to enter the api-key and Content-Type once.

For either tool, you need to choose a command (GET, POST, PUT, and so forth), provide a URL endpoint, and for some tasks, provide JSON in the body of the request. Replace the search service name (YOUR-SEARCH-SERVICE-NAME) with a valid value. Add `$select=name` to return just the name of each index.

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

Notice the HTTPS prefix, the name of the service, the name of an object (in this case, the indexes collection), and the `api-version`. The api-version is a required, lowercase string specified as `?api-version=2020-06-30` for the current version. API versions are updated regularly. Including the api-version on each request gives you full control over which one is used.

Request header composition includes two elements: `Content-Type` and the `api-key` used to authenticate to Azure Cognitive Search. Replace the admin API key (YOUR-AZURE-SEARCH-ADMIN-API-KEY) with a valid value.

```
api-key: <YOUR-AZURE-SEARCH-ADMIN-API-KEY>
Content-Type: application/json
```

In Postman, formulate a request that looks like the following screenshot. Choose **GET** as the command, provide the URL, and click **Send**. This command connects to Azure Cognitive Search, reads the indexes collection, and returns HTTP status code 200 on a successful connection. If your service has indexes already, the response will also include index definitions.

The screenshot shows a Postman request configuration. The method is set to **GET**, the URL is `https://mydemo.search.windows.net/indexes?api-version=2020-06-30`, and the **Send** button is highlighted. The **Headers** tab is selected, showing two entries: `Content-Type: application/json` and `api-key: <placeholder-api-key-for-your-service>`. Both headers have their checkboxes checked. At the bottom right, the status is shown as `Status: 200 OK` and `Time: 540 ms`.

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` index name.

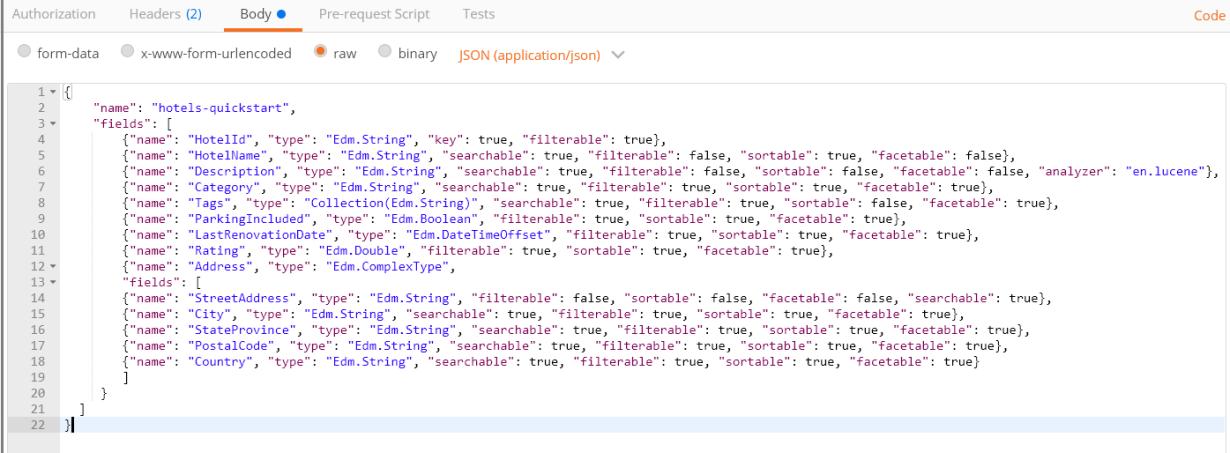
To do this in Postman:

1. Change the command to **PUT**.
2. Copy in this URL

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

3. Provide the index definition (copy-ready code is provided below) in the body of the request.

4. Click **Send**.



```
1 < [ { 2   "name": "hotels-quickstart", 3   "fields": [ 4     {"name": "HotelId", "type": "Edm.String", "key": true, "filterable": true}, 5     {"name": "HotelName", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": true, "facetable": false}, 6     {"name": "Description", "type": "Edm.String", "searchable": true, "filterable": false, "sortable": false, "facetable": false, "analyzer": "en.lucene"}, 7     {"name": "Category", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": false, "facetable": false}, 8     {"name": "Tags", "type": "Collection(Edm.String)", "searchable": true, "filterable": true, "sortable": false, "facetable": true}, 9     {"name": "ParkingIncluded", "type": "Edm.Boolean", "filterable": true, "sortable": true, "facetable": true}, 10    {"name": "LastRenovationDate", "type": "Edm.DateTimeOffset", "filterable": true, "sortable": true, "facetable": true}, 11    {"name": "Rating", "type": "Edm.Double", "filterable": true, "sortable": true, "facetable": true}, 12    {"name": "Address", "type": "Edm.ComplexType", 13      "fields": [ 14        {"name": "StreetAddress", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false, "searchable": true}, 15        {"name": "City", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true}, 16        {"name": "StateProvince", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true}, 17        {"name": "PostalCode", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true}, 18        {"name": "Country", "type": "Edm.String", "searchable": true, "filterable": true, "sortable": true, "facetable": true} 19      ] 20    } 21 22 }]
```

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 will 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.



```
{  "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": false, "facetable": false},    {"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.

To do this in Postman:

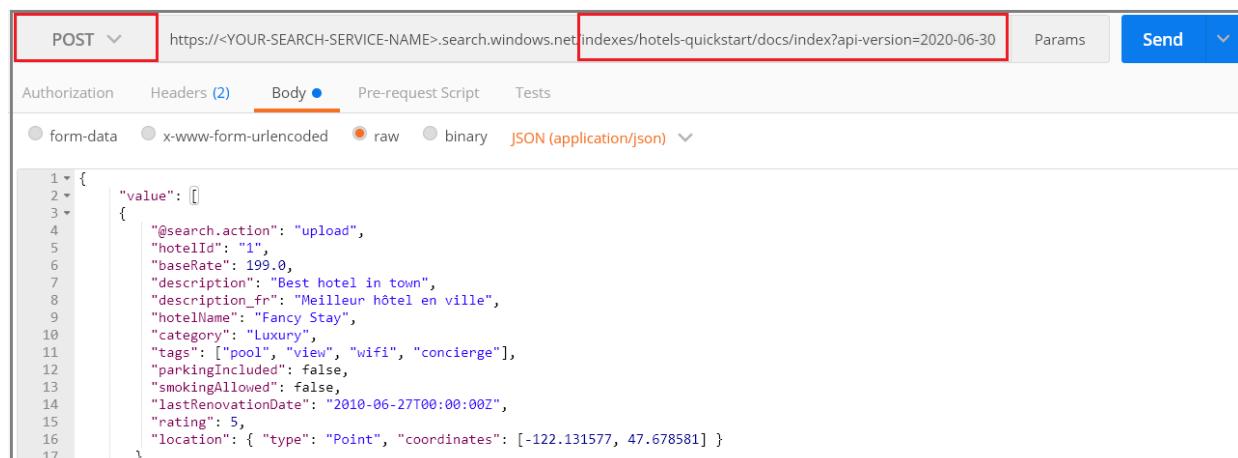
1. Change the command to POST.

2. Copy in this URL

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

3. Provide the JSON documents (copy-ready code is below) in the body of the request.

4. Click **Send**.



The screenshot shows the Postman interface with a red box highlighting the URL field: `https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/hotels-quickstart/docs/index?api-version=2020-06-30`. The **Body** tab is selected, and the `raw` option is chosen. The JSON payload is as follows:

```
1 {  
2   "value": [  
3     {  
4       "@search.action": "upload",  
5       "hotelId": "",  
6       "baseRate": 199.0,  
7       "description": "Best hotel in town",  
8       "description_fr": "Meilleur h\u00f4tel en ville",  
9       "hotelName": "Fancy Stay",  
10      "category": "Luxury",  
11      "tags": ["pool", "view", "wifi", "concierge"],  
12      "parkingIncluded": false,  
13      "smokingAllowed": false,  
14      "lastRenovationDate": "2010-06-27T00:00:00Z",  
15      "rating": 5,  
16      "location": { "type": "Point", "coordinates": [-122.131577, 47.678581] }  
17    },  
18  ]  
19}
```

JSON documents to load into the index

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

```
{  
  "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": 5  
    },  
    {  
      "@search.action": "upload",  
      "HotelId": "2",  
      "HotelName": "Grand Central Hotel",  
      "Description": "A classic New York hotel located in the heart of the city. It is close to many landmarks and offers great views of the surrounding area.",  
      "Category": "Historical",  
      "Tags": [ "central park", "midtown", "historical" ],  
      "ParkingIncluded": true,  
      "LastRenovationDate": "2018-05-15T00:00:00Z",  
      "Rating": 4  
    },  
    {  
      "@search.action": "upload",  
      "HotelId": "3",  
      "HotelName": "Times Square Hotel",  
      "Description": "A modern hotel located in the Times Square area. It is close to many attractions and offers great views of the city at night.",  
      "Category": "Entertainment",  
      "Tags": [ "times square", "theatre", "entertainment" ],  
      "ParkingIncluded": false,  
      "LastRenovationDate": "2019-10-01T00:00:00Z",  
      "Rating": 4.5  
    },  
    {  
      "@search.action": "upload",  
      "HotelId": "4",  
      "HotelName": "Metropolitan Hotel",  
      "Description": "A luxury hotel located in the heart of New York. It is close to many landmarks and offers great views of the city.",  
      "Category": "Luxury",  
      "Tags": [ "metropolitan", "luxury", "city view" ],  
      "ParkingIncluded": true,  
      "LastRenovationDate": "2020-03-01T00:00:00Z",  
      "Rating": 5  
    }  
  ]  
}
```

```
"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.

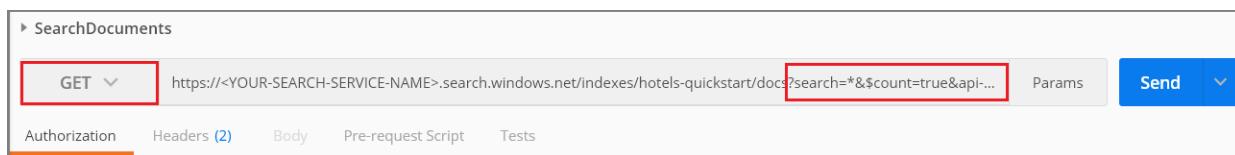
To do this in Postman:

1. Change the command to **GET**.
2. Copy in this URL

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

3. Click **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 click **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.

Swap out the current URL with the ones below, clicking **Send** each time to view the results.

```

# Query example 1 - Search on restaurant and wifi
# Return only the HotelName, Description, and Tags fields
https://<YOUR-SEARCH-SERVICE>.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 higher
# Returns the HotelName and Rating. Two documents match
https://<YOUR-SEARCH-SERVICE>.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://<YOUR-SEARCH-SERVICE>.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://<YOUR-SEARCH-SERVICE>.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:

```
https://<YOUR-SEARCH-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.

The screenshot shows a Postman interface with the following details:

- Method:** GET
- URL:** `https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/indexes/hotels-quickstart/stats?api-version=2020-06-30`
- Authorization:** (highlighted in red)
- Headers:** (2)
- Body:** (empty)
- Pre-request Script:** (empty)
- Tests:** (empty)
- Params:** (empty)
- Send:** (button)

Notice that the api-version syntax differs. For this request, use `?api-version` 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 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

Now that you know how to perform core tasks, you can move forward with additional REST API calls for more advanced features, such as indexers or [setting up a cognitive search pipeline](#). For your next step, we recommend the following link:

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

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

10/4/2020 • 10 minutes to read • [Edit Online](#)

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

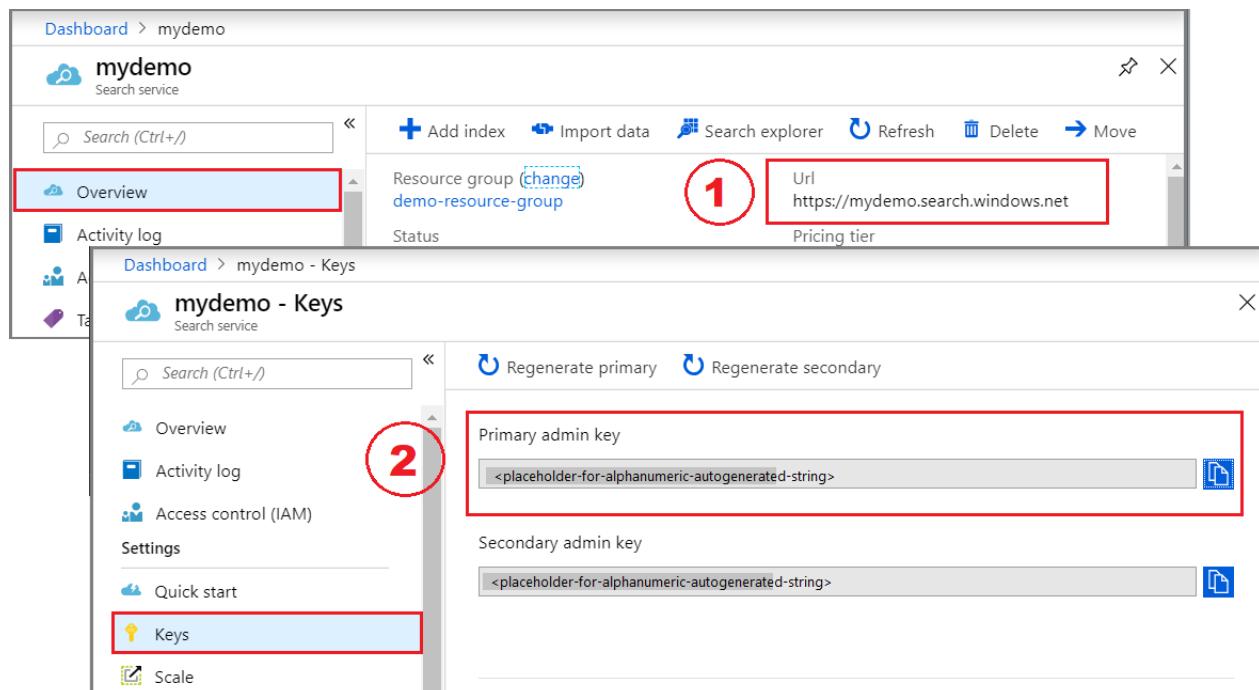
The following services and tools are required for this quickstart.

- [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.

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.

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.

```
$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.

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

```
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.

```

$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}
        ]
    }
}
"@

```

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

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

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

```
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](#).

- 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.

```
$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"
    }
}
}
"@
```

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

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

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

```
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.

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

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

```
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.

```
# 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](#)

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

10/4/2020 • 10 minutes to read • [Edit Online](#)

Build a Jupyter notebook that creates, loads, and queries an Azure Cognitive Search index using Python and the [Azure Cognitive Search REST APIs](#). 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

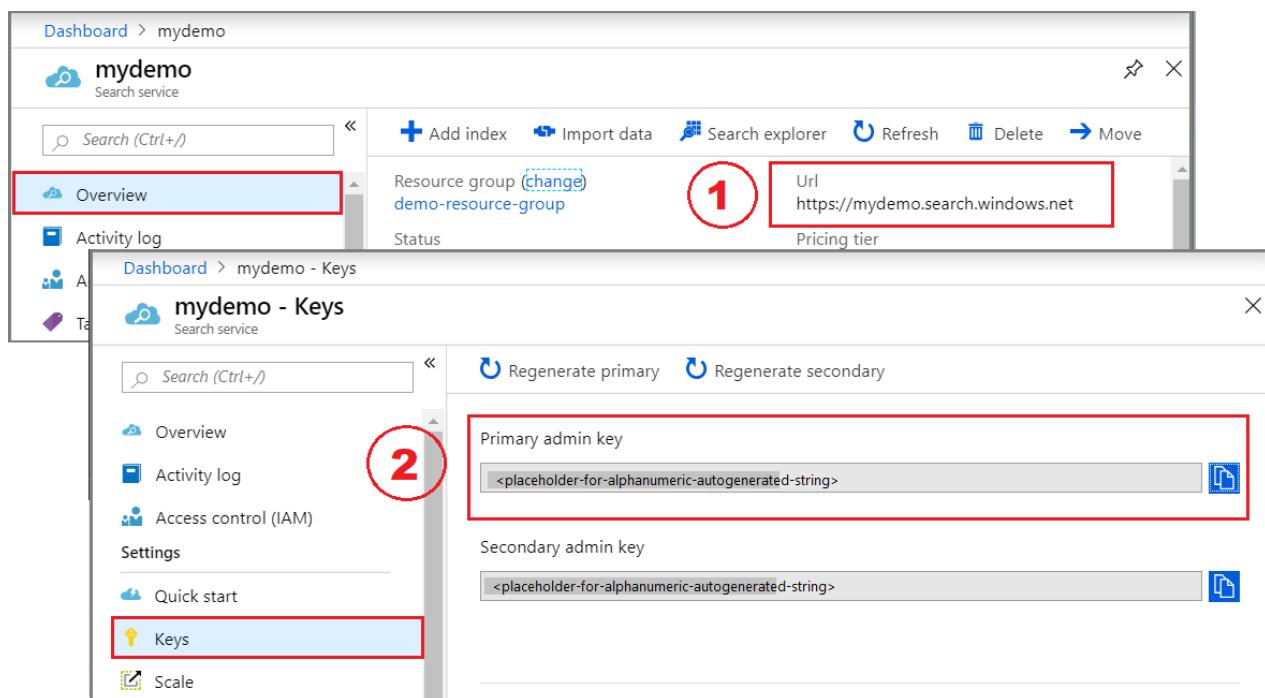
The following services and tools are required for this quickstart.

- [Anaconda 3.x](#), providing Python 3.x and Jupyter Notebooks.
- [Create an Azure Cognitive Search service](#) or [find an existing service](#) under your current subscription. You can use the Free tier for this quickstart.

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.

Connect to Azure Cognitive Search

In this task, start a Jupyter notebook and verify that you can connect to Azure Cognitive Search. You'll do this by requesting a list of indexes from your service. On Windows with Anaconda3, you can use Anaconda Navigator to launch a notebook.

1. Create a new Python3 notebook.
2. In the first cell, load the libraries used for working with JSON and formulating HTTP requests.

```
import json
import requests
from pprint import pprint
```

3. In the second cell, input the request elements that will be constants on every request. Replace the search service name (YOUR-SEARCH-SERVICE-NAME) and admin API key (YOUR-ADMIN-API-KEY) with valid values.

```
endpoint = 'https://<YOUR-SEARCH-SERVICE-NAME>.search.windows.net/'
api_version = '?api-version=2020-06-30'
headers = {'Content-Type': 'application/json',
           'api-key': '<YOUR-ADMIN-API-KEY>' }
```

If you get ConnectionError "Failed to establish a new connection", verify that the api-key is a primary or secondary admin key, and that all leading and trailing characters (? and /) are in place.

4. In the third cell, formulate the request. This GET request targets the indexes collection of your search service and selects the name property of existing indexes.

```
url = endpoint + "indexes" + api_version + "&$select=name"
response = requests.get(url, headers=headers)
index_list = response.json()
pprint(index_list)
```

5. Run each step. If indexes exist, the response contains a list of index names. In the screenshot below, the service already has an azureblob-index and a realestate-us-sample index.

```
In [1]: import json
        import requests
        from pprint import pprint

In [2]: endpoint = 'https://mydemo.search.windows.net/'
        api_version = '?api-version=2019-05-06'
        headers = {'Content-Type': 'application/json',
                   'api-key': 'AA11BB22CC33DD44FF55EE66GG77HH88II99' }

In [3]: url = endpoint + "indexes" + api_version + "&$select=name"
        response = requests.get(url, headers=headers)
        index_list = response.json()
        pprint(index_list)

{'@odata.context': 'https://mydemo.search.windows.net/$metadata#indexes(name)',
 'value': [{'name': 'azureblob-index'}, {'name': 'realestate-us-sample'}]}
```

In contrast, an empty index collection returns this response:

```
{'@odata.context': 'https://mydemo.search.windows.net/$metadata#indexes(name)', '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 uses the [Create Index REST API](#) to push an index schema to the service.

Required elements of an index include a name, a fields collection, and a key. The fields collection defines the structure of a *document*. Each field has a name, type, and attributes that determine how the field is 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 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.

```
index_schema = {
    "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": "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": "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. In another cell, formulate the request. This POST request targets the indexes collection of your search service and creates an index based on the index schema you provided in the previous cell.

```
url = endpoint + "indexes" + api_version
response = requests.post(url, headers=headers, json=index_schema)
index = response.json()
pprint(index)
```

3. Run each step.

The response includes the JSON representation of the schema. The following screenshot is showing just a portion of the response.

```
In [4]: index_schema = {
    "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": "true"}, 
        {"name": "Description", "type": "Edm.String", "searchable": "true", "filterable": "false", "sortable": "false", "facetable": "true"}, 
        {"name": "Description_fr", "type": "Edm.String", "searchable": "true", "filterable": "false", "sortable": "false", "facetable": "true"}, 
        {"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.Decimal", "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"} 
        ] 
    ] 
}
```

```
In [5]: url = endpoint + "indexes" + api_version
response = requests.post(url, headers=headers, json=index_schema)
index = response.json()
pprint(index)
```

```
{'@odata.context': 'https://mydemo.search.windows.net/$metadata#indexes$/entity',
 '@odata.etag': '"0x8D6EDC5DE036D1A"',
 'analyzers': [],
 'charFilters': [],
 'corsOptions': None,
 'defaultScoringProfile': None,
 'fields': [{'analyzer': None,
             'facetable': True,
             'filterable': True,
             'indexAnalyzer': None,
             'key': True,
             'name': 'HotelId',
             'retrievable': True,
             'searchAnalyzer': None,
             'searchable': True,
             'sortable': True}]}>
```

TIP

Another way to verify index creation is to check the Indexes list in the portal.

2 - Load documents

To push documents, use an HTTP POST request to your index's URL endpoint. The REST API is [Add, Update, or Delete Documents](#). 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.

```
documents = {
    "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.",
            "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": "10019",
                "Country": "USA"
            }
        }
    ]
}
```

```
        "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 POST request targets the docs collection of the hotels-quickstart index and pushes the documents provided in the previous step.

```

url = endpoint + "indexes/hotels-quickstart/docs/index" + api_version
response = requests.post(url, headers=headers, json=documents)
index_content = response.json()
pprint(index_content)

```

3. Run each step to push the documents to an index in your search service. Results should look similar to the following example.

```

In [6]: url = endpoint + "indexes/hotels-quickstart/docs/index" + api_version
response = requests.post(url, headers=headers, json=documents)
index_content = response.json()
pprint(index_content)

{'@odata.context': "https://mydemo.search.windows.net/indexes('hotels-quickstart')/$metadata#Collection(Microsoft.Azure.Sea
rch.V2019_05_06.IndexResult)",
'value': [{"errorMessage': None,
'key': '1',
'status': True,
'statusCode': 201},
{'errorMessage': None,
'key': '2',
'status': True,
'statusCode': 201},
{'errorMessage': None,
'key': '3',
'status': True,
'statusCode': 201},
{'errorMessage': None,
'key': '4',
'status': True,
'statusCode': 201}]}

```

3 - Search an index

This step shows you how to query an index using the [Search Documents REST API](#).

1. In a cell, provide a query expression that 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.

```

searchstring = '&search=*&$count=true'

url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
response = requests.get(url, headers=headers, json=searchstring)
query = response.json()
pprint(query)

```

2. In a new cell, provide the following example to search on the terms "hotels" and "wifi". Add \$select to specify which fields to include in the search results.

```

searchstring = '&search=hotels wifi&$count=true&$select=HotelId,HotelName'

url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
response = requests.get(url, headers=headers, json=searchstring)
query = response.json()
pprint(query)

```

Results should look similar to the following output.

```

In [11]: M searchstring = '&search=hotels wifi&$count=true&$select=HotelId,HotelName'

In [12]: M url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
          response = requests.get(url, headers=headers, json=searchstring)
          query = response.json()
          pprint(query)

{'@odata.context': "https://mydemo.search.windows.net/indexes('hotels-quickstart')/$metadata#docs(*)",
 '@odata.count': 4,
 'value': [{"@search.score": 0.16548625,
            'HotelId': '2',
            'HotelName': 'Twin Dome Motel'},
           {"@search.score": 0.016361875,
            'HotelId': '3',
            'HotelName': 'Triple Landscape Hotel'},
           {"@search.score": 0.008899688,
            'HotelId': '1',
            'HotelName': 'Secret Point Motel'},
           {"@search.score": 0.008899688,
            'HotelId': '4',
            'HotelName': 'Sublime Cliff Hotel'}]}

```

3. Next, apply a \$filter expression that selects only those hotels with a rating greater than 4.

```

searchstring = '&search=*&$filter=Rating gt 4&$select=HotelId,HotelName,Description,Rating'

url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
response = requests.get(url, headers=headers, json=searchstring)
query = response.json()
pprint(query)

```

4. By default, the search engine returns the top 50 documents but you can use top and skip to add pagination and choose how many documents in each result. This query returns two documents in each result set.

```

searchstring = '&search=boutique&$top=2&$select=HotelId,HotelName,Description'

url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
response = requests.get(url, headers=headers, json=searchstring)
query = response.json()
pprint(query)

```

5. In this last example, use \$orderby to sort results by city. This example includes fields from the Address collection.

```

searchstring = '&search=pool&$orderby=Address/City&$select=HotelId, HotelName, Address/City,
Address/StateProvince'

url = endpoint + "indexes/hotels-quickstart/docs" + api_version + searchstring
response = requests.get(url, headers=headers, json=searchstring)
query = response.json()
pprint(query)

```

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.

Next steps

As a simplification, this quickstart uses an abbreviated version of the Hotels index. You can create the full version to try out more interesting queries. To get the full version and all 50 documents, run the **Import data** wizard, selecting *hotels-sample* from the built-in sample data sources.

[Quickstart: Create an index in the Azure portal](#)

Tutorial: Create your first search app using the .NET SDK

10/4/2020 • 14 minutes to read • [Edit Online](#)

This tutorial shows you how to create a web app that queries and returns results from a search index using Azure Cognitive Search and Visual Studio.

In this tutorial, you will learn how to:

- Setup a development environment
- Model data structures
- Create a web page to collect query inputs and display results
- Define a search method
- Test the app

You will also learn how straightforward a search call is. The key statements in the code you will develop are encapsulated in the following few lines.

```
var options = new SearchOptions()
{
    // The Select option specifies fields for the result set
    options.Select.Add("HotelName");
    options.Select.Add("Description");
};

var searchResult = await _searchClient.SearchAsync<Hotel>(model.searchText, options).ConfigureAwait(false);
model.resultList = searchResult.Value.GetResults().ToList();
```

Just one call queries the index and returns results.



16 Results

Travel Resort

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.

Nova Hotel & Spa

1 Mile from the airport. Free WiFi, Outdoor Pool, Complimentary Airport Shuttle, 6 miles from the beach & 10 miles from downtown.

King's Palace Hotel

Newest kid on the downtown block. Steps away from the most popular destinations in downtown, enjoy free WiFi, an indoor rooftop pool & fitness center, 24 Grab'n'Go & drinks at the bar

Overview

This tutorial uses an existing, hosted sample index so that you can focus on building a search page that collects a query string for the request and returns results. The index contains fictitious hotel data. Once you have a basic page, you can enhance it in subsequent lessons to include paging, facets, and a type-ahead experience.

A finished version of the code in this tutorial can be found in the following project:

- [1-basic-search-page \(GitHub\)](#)

Prerequisites

- [Visual Studio](#)
- [Azure Cognitive Search client library \(version 11\)](#)

This tutorial has been updated to use the Azure.Search.Documents (version 11) package. For an earlier version of the .NET SDK, see [Microsoft.Azure.Search \(version 10\) code sample](#).

Because you are using a public sample hosted by Microsoft, you don't need a search service or an Azure account for this tutorial.

Install and run the project from GitHub

If you want to jump ahead to a working app, follow the steps below to download and run the finished code.

1. Locate the sample on GitHub: [Create first app](#).
2. At the [root folder](#), select **Code**, followed by **Clone** or **Download ZIP** to make your private local copy of the project.

3. Using Visual Studio, navigate to, and open the solution for the basic search page ("1-basic-search-page"), and select **Start without debugging** (or press F5) to build and run the program.
4. This is a hotels index, so type in some words that you might use to search for hotels (for example, "wifi", "view", "bar", "parking"), and examine the results.

The screenshot shows a web application titled "Hotels Search". At the top, there is a search bar with the word "wifi" typed into it, and a magnifying glass icon to its right. Below the search bar, the text "19 Results" is displayed. The results are presented in four cards, each representing a different hotel:

- Super Deluxe Inn & Suites**: Described as a Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor/Outdoor Pool, Fitness Center, Florida Green certified, Starbucks Coffee, HDTV.
- Double Sanctuary Resort**: Described as a 5* Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Conde Nast Traveler. Free WiFi, Flexible check in/out, Fitness Center & Nespresso in room.
- Veteran Right Track**: Described as offering Free Shuttle to the airport and casinos. Free breakfast and WiFi.
- Regal Orb Resort & Spa**: Described as Your home away from home. Brand new fully equipped premium rooms, fast WiFi, full kitchen, washer & dryer, fitness center.

Hopefully this project will run smoothly, and you have Web app running. Many of the essential components for more sophisticated searches are included in this one app, so it is a good idea to go through it, and recreate it step by step. The following sections cover these steps.

Set up a development environment

To create this project from scratch, and thus reinforce the concepts of Azure Cognitive Search in your mind, start with a Visual Studio project.

1. In Visual Studio, select **New > Project**, then **ASP.NET Core Web Application**.

Create a new project

Recent project templates

The screenshot shows the 'Create a new project' dialog with a search bar at the top. Below it, there are dropdown menus for 'All languages', 'All platforms', and 'All project types'. A list of recent templates is shown on the left, including:

- Console App (.NET Core) - C#
- ASP.NET Web Application
- Application (.NET Framework) - Visual Basic
- ASP.NET Core Web Application - C#
- Blank Node.js Console Application - JavaScript
- Blank Node.js Web Application - JavaScript

A red box highlights the 'ASP.NET Core Web Application' template, which is described as follows:

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.

Below the template description are buttons for C#, Linux, macOS, Windows, Cloud, Service, and Web.

Next

2. Give the project a name such as "FirstSearchApp" and set the location. Select **Create**.

3. Choose the **Web Application (Model-View-Controller)** project template.

Create a new ASP.NET Core web application

The screenshot shows the 'Create a new ASP.NET Core web application' dialog. At the top, there are dropdown menus for '.NET Core' (set to 'ASP.NET Core 3.1') and 'Authentication' (set to 'No Authentication').

The left side lists several project templates:

- Empty**: An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
- API**: A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
- Web Application**: A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.
- Web Application (Model-View-Controller)**: A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services. This template is highlighted with a red box.
- Angular**: A project template for creating an ASP.NET Core application with Angular.
- React.js**: A project template for creating an ASP.NET Core application with React.js.

The right side of the dialog contains settings for 'Advanced' features:

- Configure for HTTPS
- Enable Docker Support
(Requires Docker Desktop)
- Enable Razor runtime compilation

At the bottom, there are buttons for 'Get additional project templates', 'Back', and 'Create'.

4. Install the client library. In **Tools > NuGet Package Manager > Manage NuGet Packages for Solution...**, select **Browse** and then search for "azure.search.documents". Install **Azure.Search.Documents** (version 11 or later), accepting the license agreements and dependencies.

NuGet - Solution FirstSearchApp

Browse Installed Updates Consolidate

Manage Packages for Solution

Package source: nuget.org

azuresearch.documents

Azure.Search.Documents by Microsoft, v11.1.1
This is the Azure Cognitive Search client library for developing .NET

Microsoft.Azure.Search.Service by Microsoft, v10.1.0
Use this package if you're developing automation in .NET to manage Azure Cognitive Search index...

Microsoft.Azure.Search.Data by Microsoft, v10.1.0
Use this package if you're developing a .NET application using Azure Cognitive Search, and y...

Citolab.Azure.BlobStorage.Document v2020.1.29.2
Helper to upload documents to Azure blob storage. Makes it easy to do full text search...

Citolab.Azure.BlobStorage.Document v2020.1.29.2
Helper to upload documents to Azure blob storage. Makes it easy to do full text search...

Microsoft.Azure.Management.Search v1.34.0
Provides search service accounts management

Versions - 0

Project	Version
FirstSearchApp	v11.1.1

Installed: not installed Uninstall

Version: Latest stable 11.1.1 Install

Options

Description

Initialize Azure Cognitive Search

For this sample, you are using publicly available hotel data. This data is an arbitrary collection of 50 fictional hotel names and descriptions, created solely for the purpose of providing demo data. To access this data, specify a name and API key.

1. Open `appsettings.json` and replace the default lines with the following name and key. The API key shown here is not an example of a key, it is *exactly* the key you need to access the hotel data. Your file should now look like this.

```
{
  "SearchServiceName": "azs-playground",
  "SearchServiceQueryApiKey": "EA4510A6219E14888741FCFC19BFBB82"
}
```

2. In Solution Explorer, select the file, and in Properties, change the **Copy to Output Directory** setting to **Copy if newer**.

Properties

appsettings.json File Properties

Advanced

Build Action	Content
Copy to Output Director	Copy if newer
Custom Tool	
Custom Tool Namespace	

Misc

File Name	appsettings.json
-----------	------------------

Model data structures

Models (C# classes) are used to communicate data between the client (the view), the server (the controller), and

also the Azure cloud using the MVC (model, view, controller) architecture. Typically, these models will reflect the structure of the data that is being accessed.

In this step, you'll model the data structures of the search index, as well as the search string used in view/controller communications. In the hotels index, each hotel has many rooms, and each hotel has a multi-part address. Altogether, the full representation of a hotel is a hierarchical and nested data structure. You will need three classes to create each component.

The set of **Hotel**, **Address**, and **Room** classes are known as *complex types*, an important feature of Azure Cognitive Search. Complex types can be many levels deep of classes and subclasses, and enable far more complex data structures to be represented than using *simple types* (a class containing only primitive members).

1. In Solution Explorer, right-click **Models** > **Add** > **New Item**.
2. Select **Class** and name the item **Hotel.cs**. Replace all the contents of **Hotel.cs** with the following code. Notice the **Address** and **Room** members of the class, these fields are classes themselves so you will need models for them too.

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

namespace FirstAzureSearchApp.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 Room[] Rooms { get; set; }
    }
}
```

3. Repeat the same process of creating a model for the **Address** class, naming the file Address.cs. Replace the contents with the following.

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

namespace FirstAzureSearchApp.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; }
    }
}
```

4. And again, follow the same process to create the **Room** class, naming the file Room.cs.

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

namespace FirstAzureSearchApp.Models
{
    public partial class Room
    {
        [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; }
    }
}
```

5. The last model you'll create in this tutorial is a class named **SearchData** and it represents the user's input

(**searchText**), and the search's output (**resultList**). The type of the output is critical, **SearchResults<Hotel>**, as this type exactly matches the results from the search, and you need to pass this reference through to the view. Replace the default template with the following code.

```
using Azure.Search.Documents.Models;

namespace FirstAzureSearchApp.Models
{
    public class searchData
    {
        // The text to search for.
        public string searchText { get; set; }

        // The list of results.
        public SearchResults<Hotel> resultList;
    }
}
```

Create a web page

Project templates come with a number of client views located in the **Views** folder. The exact views depend on the version of Core .NET you are using (3.1 is used in this sample). For this tutorial, you will modify **Index.cshtml** to include the elements of a search page.

Delete the content of Index.cshtml in its entirety, and rebuild the file in the following steps.

1. The tutorial uses two small images in the view: an Azure logo and a search magnifier icon (azure-logo.png and search.png). Copy across the images from the GitHub project to the **wwwroot/images** folder in your project.
2. The first line of Index.cshtml should reference the model used to communicate data between the client (the view) and the server (the controller), which is the **SearchData** model previously created. Add this line to the Index.cshtml file.

```
@model FirstAzureSearchApp.Models.SearchData
```

3. It is standard practice to enter a title for the view, so the next lines should be:

```
@{
    ViewData["Title"] = "Home Page";
}
```

4. Following the title, enter a reference to an HTML stylesheet, which you will create shortly.

```
<head>
    <link rel="stylesheet" href="~/css/hotels.css" />
</head>
```

5. The body of the view handles two use cases. First, it must provide an empty page on first use, before any search text is entered. Secondly, it must handle results, in addition to the search text box, for repeated queries.

To handle both cases, you need to check whether the model provided to the view is null. A null model indicates the first use case (the initial running of the app). Add the following to the Index.cshtml file and read through the comments.

```

<body>
<h1 class="sampleTitle">
    <img src("~/images/azure-logo.png" width="80" />
    Hotels Search
</h1>

@using (Html.BeginForm("Index", "Home", FormMethod.Post))
{
    // Display the search text box, with the search icon to the right of it.
    <div class="searchBoxForm">
        @Html.TextBoxFor(m => m.searchText, new { @class = "searchBox" }) <input
        class="searchBoxSubmit" type="submit" value="">
    </div>

    @if (Model != null)
    {
        // Show the result count.
        <p class="sampleText">
            @Model.resultList.Count Results
        </p>

        @for (var i = 0; i < Model.resultList.Count; i++)
        {
            // Display the hotel name and description.
            @Html.TextAreaFor(m => m.resultList[i].Document.HotelName, new { @class = "box1" })
            @Html.TextArea($"desc{i}", Model.resultList[i].Document.Description, new { @class = "box2" })
        }
    }
}
</body>

```

6. Add the stylesheet. In Visual Studio, in **File > New > File**, select **Style Sheet** (with **General** highlighted).

Replace the default code with the following. We will not be going into this file in any more detail, the styles are standard HTML.

```

textarea.box1 {
    width: 648px;
    height: 30px;
    border: none;
    background-color: azure;
    font-size: 14pt;
    color: blue;
    padding-left: 5px;
}

textarea.box2 {
    width: 648px;
    height: 100px;
    border: none;
    background-color: azure;
    font-size: 12pt;
    padding-left: 5px;
    margin-bottom: 24px;
}

.sampleTitle {
    font: 32px/normal 'Segoe UI Light', Arial, Helvetica, Sans-Serif;
    margin: 20px 0;
    font-size: 32px;
    text-align: left;
}

.sampleText {
    font: 16px/bold 'Segoe UI Light', Arial, Helvetica, Sans-Serif;
}

```

```

        font-size: 14px;
        text-align: left;
        height: 30px;
    }

.searchBoxForm {
    width: 648px;
    box-shadow: 0 0 0 1px rgba(0,0,0,.1), 0 2px 4px 0 rgba(0,0,0,.16);
    background-color: #fff;
    display: inline-block;
    border-collapse: collapse;
    border-spacing: 0;
    list-style: none;
    color: #666;
}

.searchBox {
    width: 568px;
    font-size: 16px;
    margin: 5px 0 1px 20px;
    padding: 0 10px 0 0;
    border: 0;
    max-height: 30px;
    outline: none;
    box-sizing: content-box;
    height: 35px;
    vertical-align: top;
}

.searchBoxSubmit {
    background-color: #fff;
    border-color: #fff;
    background-image: url(/images/search.png);
    background-repeat: no-repeat;
    height: 20px;
    width: 20px;
    text-indent: -99em;
    border-width: 0;
    border-style: solid;
    margin: 10px;
    outline: 0;
}

```

7. Save the stylesheet file as `hotels.css`, into the `wwwroot/css` folder, alongside the default `site.css` file.

That completes our view. At this point, both the models and views are completed. Only the controller is left to tie everything together.

Define methods

In this step, modify to the contents of **Home Controller**.

1. Open the `HomeController.cs` file and replace the `using` statements with the following.

```
using Azure;
using Azure.Search.Documents;
using Azure.Search.Documents.Indexes;
using FirstAzureSearchApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
```

Add Index methods

In an MVC app, the `Index()` method is a default action method for any controller. It opens the index HTML page. The default method, which takes no parameters, is used in this tutorial for the application start-up use case: rendering an empty search page.

In this section, we extend the method to support a second use case: rendering the page when a user has entered search text. To support this case, the index method is extended to take a model as a parameter.

1. Add the following method, after the default `Index()` method.

```
[HttpPost]
public async Task<ActionResult> Index(SearchData model)
{
    try
    {
        // Ensure the search string is valid.
        if (model.searchText == null)
        {
            model.searchText = "";
        }

        // Make the Azure Cognitive Search call.
        await RunQueryAsync(model);
    }

    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "1" });
    }
    return View(model);
}
```

Notice the `async` declaration of the method, and the `await` call to `RunQueryAsync`. These keywords take care of making asynchronous calls, and thus avoid blocking threads on the server.

The `catch` block uses the error model that was created by default.

Note the error handling and other default views and methods

Depending on which version of .NET Core you are using, a slightly different set of default views are created by default. For .NET Core 3.1 the default views are Index, Privacy, and Error. You can view these default pages when running the app, and examine how they are handled in the controller.

You will be testing the Error view later on in this tutorial.

In the GitHub sample, unused views and their associated actions are deleted.

Add the `RunQueryAsync` method

The Azure Cognitive Search call is encapsulated in our `RunQueryAsync` method.

1. First add some static variables to set up the Azure service, and a call to initiate them.

```
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");
}
```

2. Now, add the **RunQueryAsync** method itself.

```
private async Task<ActionResult> RunQueryAsync(SearchData model)
{
    InitSearch();

    var options = new SearchOptions() { };

    // Enter Hotel property names into this list so only these values will be returned.
    // If Select is empty, all values will be returned, which can be inefficient.
    options.Select.Add("HotelName");
    options.Select.Add("Description");

    // For efficiency, the search call should be asynchronous, so use SearchAsync rather than Search.
    var searchResult = await _searchClient.SearchAsync<Hotel>(model.searchText,
options).ConfigureAwait(false);
    model.resultList = searchResult.Value.GetResults().ToList();

    // Display the results.
    return View("Index", model);
}
```

In this method, first ensure our Azure configuration is initiated, then set some search options. The **Select** option specifies which fields to return in results, and thus match the property names in the **Hotel** class. If you omit **Select**, all unhidden fields are returned, which can be inefficient if you are only interested in a subset of all possible fields.

The asynchronous call to search formulates the request (modeled as **searchText**) and response (modeled as **searchResult**). If you are debugging this code, the **SearchResult** class is a good candidate for setting a break point if you need to examine the contents of **model.resultList**. You should find that it is intuitive, providing you with the data you asked for, and not much else.

Test the app

Now, let's check whether the app runs correctly.

1. Select **Debug > Start Without Debugging** or press F5. If the app runs as expected, you should get the initial Index view.



Hotels Search



2. Enter a query string such as "beach" (or any text that comes to mind), and click the search icon to send the request.

Hotels Search

beach

6 Results

Whitefish Lodge & Suites
Located on in the heart of the forest. Enjoy Warm Weather, Beach Club Services, Natural Hot Springs, Airport Shuttle.

Lady Of The Lake B & B
Nature is Home on the beach. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackout may apply.

Nova Hotel & Spa
1 Mile from the airport. Free WiFi, Outdoor Pool, Complimentary Airport Shuttle, 6 miles from the beach & 10 miles from downtown.

Ocean Air Motel
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.

3. Try entering "five star". Notice that this query returns no results. A more sophisticated search would treat "five star" as a synonym for "luxury" and return those results. Support for [synonyms](#) is available in Azure Cognitive Search, but isn't be covered in this tutorial series.
4. Try entering "hot" as search text. It does *not* return entries with the word "hotel" in them. Our search is only locating whole words, though a few results are returned.
5. Try other words: "pool", "sunshine", "view", and whatever. You will see Azure Cognitive Search working at its simplest, but still convincing level.

Test edge conditions and errors

It is important to verify that our error handling features work as they should, even when things are working perfectly.

1. In the `Index` method, after the `try {` call, enter the line `Throw new Exception()`. This exception will force an error when you search on text.
2. Run the app, enter "bar" as search text, and click the search icon. The exception should result in the error view.

Error.

An error occurred while processing your request.

Request ID: 1

Development Mode

Switching to **Development** environment will display more detailed information about the error that occurred.

Development environment should not be enabled in deployed applications, as it can result in sensitive information from exceptions being displayed to end users. For local debugging, development environment can be enabled by setting the `ASPNETCORE_ENVIRONMENT` environment variable to **Development**, and restarting the application.

© 2019 - FirstAzureSearchApp

IMPORTANT

It is considered a security risk to return internal error numbers in error pages. If your app is intended for general use, do some investigation into secure and best practices of what to return when an error occurs.

3. Remove `Throw new Exception()` when you are satisfied the error handling works as it should.

Takeaways

Consider the following takeaways from this project:

- An Azure Cognitive Search call is concise, and it is easy to interpret the results.
- Asynchronous calls add a small amount of complexity to the controller, but are the best practice if you intend to develop quality apps.
- This app performed a straightforward text search, defined by what is set up in `searchOptions`. However, this one class can be populated with many members that add sophistication to a search. Not much additional work is needed to make this app considerably more powerful.

Next steps

To improve upon the user experience, add more features, notably paging (either using page numbers, or infinite scrolling), and autocomplete/suggestions. You can also consider more sophisticated search options (for example, geographical searches on hotels within a specified radius of a given point, and search results ordering).

These next steps are addressed in the remaining tutorials. Let's start with paging.

[C# Tutorial: Search results pagination - Azure Cognitive Search](#)

Tutorial: Add paging to search results using the .NET SDK

10/4/2020 • 16 minutes to read • [Edit Online](#)

Learn how to implement two different paging systems, the first based on page numbers and the second on infinite scrolling. Both systems of paging are widely used, and selecting the right one depends on the user experience you would like with the results.

In this tutorial, learn how to:

- Extend your app with numbered paging
- Extend your app with infinite scrolling

Overview

This tutorial overlays a paging system into a previously created project described in the [Create your first search app](#) tutorial.

Finished versions of the code that you will develop in this tutorial can be found in the following projects:

- [2a-add-paging \(GitHub\)](#)
- [2b-add-infinite-scroll \(GitHub\)](#)

Prerequisites

- [1-basic-search-page \(GitHub\)](#) project. This project can either be your own version built from the previous tutorial or a copy from GitHub.

This tutorial has been updated to use the [Azure.Search.Documents \(version 11\)](#) package. For an earlier version of the .NET SDK, see [Microsoft.Azure.Search \(version 10\) code sample](#).

Extend your app with numbered paging

Numbered paging is the paging system of choice for the main commercial web search engines and many other search websites. Numbered paging typically includes a "next" and "previous" option in addition to a range of actual page numbers. Also a "first page" and "last page" option might also be available. These options certainly give a user control over navigating through page-based results.

In this tutorial, you will add a system that includes first, previous, next, and last options, along with page numbers that do not start from 1, but instead surround the current page the user is on (so, for example, if the user is looking at page 10, perhaps page numbers 8, 9, 10, 11, and 12 are displayed).

The system will be flexible enough to allow the number of visible page numbers to be set in a global variable.

The system will treat the left-most and right-most page number buttons as special, meaning they will trigger changing the range of page numbers displayed. For example, if page numbers 8, 9, 10, 11 and 12 are displayed, and the user clicks on 8, then the range of page numbers displayed changes to 6, 7, 8, 9, and 10. And there is a similar shift to the right if they selected 12.

Add paging fields to the model

Have the basic search page solution open.

1. Open the SearchData.cs model file.
2. Add global variables to support pagination. In MVC, global variables are declared in their own static class. **ResultsPerPage** sets the number of results per page. **MaxPageRange** determines the number of visible page numbers on the view. **PageRangeDelta** determines how many pages should be shifted left or right, when the left-most or right-most page number is selected. Typically this latter number is around half of **MaxPageRange**. Add the following code into the namespace.

```
public static class GlobalVariables
{
    public static int ResultsPerPage
    {
        get
        {
            return 3;
        }
    }

    public static int MaxPageRange
    {
        get
        {
            return 5;
        }
    }

    public static int PageRangeDelta
    {
        get
        {
            return 2;
        }
    }
}
```

TIP

If you are running this project on a device with a smaller screen, such as a laptop, then consider changing **ResultsPerPage** to 2.

3. Add paging properties to the **SearchData** class, after the **searchText** property.

```
// The current page being displayed.
public int currentPage { get; set; }

// The total number of pages of results.
public int pageCount { get; set; }

// The left-most page number to display.
public int leftMostPage { get; set; }

// The number of page numbers to display - which can be less than MaxPageRange towards the end of the results.
public int pageRange { get; set; }

// Used when page numbers, or next or prev buttons, have been selected.
public string paging { get; set; }
```

Add a table of paging options to the view

1. Open the index.cshtml file, and add the following code right before the closing </body> tag. This new code presents a table of paging options: first, previous, 1, 2, 3, 4, 5, next, last.

```

@if (Model != null && Model.pageCount > 1)
{
// If there is more than one page of results, show the paging buttons.


|                                                                                                                                                                                                           |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| @if (Model.currentPage > 0) {     <p class="pageButton">         @Html.ActionLink(" <", "Page", "Home", new { paging = "0" }, null)     </p> } else {     <p class="pageButtonDisabled"> &lt;{/p> }       |
| @if (Model.currentPage > 0) {     <p class="pageButton">         @Html.ActionLink("<", "PageAsync", "Home", new { paging = "prev" }, null)     </p> } else {     <p class="pageButtonDisabled">&lt;{/p> } |


```

```

        </p>
    }
    else
    {
        <p class="pageButtonDisabled">&gt;|</p>
    }
</td>
</tr>
</table>
}

```

We use an HTML table to align things neatly. However all the action comes from the `@Html.ActionLink` statements, each calling the controller with a `new` model created with different entries to the `paging` property we added earlier.

The first and last page options do not send strings such as "first" and "last", but instead send the correct page numbers.

2. Add paging classes to the list of HTML styles in the `hotels.css` file. The `pageSelected` class is there to identify the current page (by applying a bold format to the page number) in the list of page numbers.

```

.pageButton {
    border: none;
    color: darkblue;
    font-weight: normal;
    width: 50px;
}

.pageSelected {
    border: none;
    color: black;
    font-weight: bold;
    width: 50px;
}

.pageButtonDisabled {
    border: none;
    color: lightgray;
    font-weight: bold;
    width: 50px;
}

```

Add a Page action to the controller

1. Open the `HomeController.cs` file, and add the `PageAsync` action. This action responds to any of the page options selected.

```

public async Task<ActionResult> PageAsync(SearchData model)
{
    try
    {
        int page;

        switch (model.paging)
        {
            case "prev":
                page = (int)TempData["page"] - 1;
                break;

            case "next":
                page = (int)TempData["page"] + 1;
                break;

            default:
                page = int.Parse(model.paging);
                break;
        }

        // Recover the leftMostPage.
        int leftMostPage = (int)TempData["leftMostPage"];

        // Recover the search text and search for the data for the new page.
        model.searchText = TempData["searchfor"].ToString();

        await RunQueryAsync(model, page, leftMostPage);

        // Ensure TempData is stored for next call, as TempData only stores for one call.
        TempData["page"] = (object)page;
        TempData["searchfor"] = model.searchText;
        TempData["leftMostPage"] = model.leftMostPage;
    }

    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "2" });
    }
    return View("Index", model);
}

```

The `RunQueryAsync` method will now show a syntax error, because of the third parameter, which we will come to in a bit.

NOTE

The calls to `TempData` store a value (an `object`) in temporary storage, though this storage persists for *only* one call. If we store something in temporary data, it will be available for the next call to a controller action, but will most definitely be gone by the call after that. Because of this short lifespan, we store the search text and paging properties back in temporary storage each and every call to `PageAsync`.

2. Update the `Index(model)` action to store temporary variables, and to add the left-most page parameter to the `RunQueryAsync` call.

```
public async Task<ActionResult> Index(SearchData model)
{
    try
    {
        // Ensure the search string is valid.
        if (model.searchText == null)
        {
            model.searchText = "";
        }

        // Make the search call for the first page.
        await RunQueryAsync(model, 0, 0);

        // Ensure temporary data is stored for the next call.
        TempData["page"] = 0;
        TempData["leftMostPage"] = 0;
        TempData["searchfor"] = model.searchText;
    }

    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "1" });
    }
    return View(model);
}
```

3. The `RunQueryAsync` method, introduced in the previous lesson, needs modification to resolve the syntax error. We use the `Skip`, `Size`, and `IncludeTotalCount` fields of the `SearchOptions` class to request only one page worth of results, starting at the `Skip` setting. We also need to calculate the paging variables for our view. Replace the entire method with the following code.

```

private async Task<ActionResult> RunQueryAsync(SearchData model, int page, int leftMostPage)
{
    InitSearch();

    var options = new SearchOptions
    {
        // Skip past results that have already been returned.
        Skip = page * GlobalVariables.ResultsPerPage,

        // Take only the next page worth of results.
        Size = GlobalVariables.ResultsPerPage,

        // Include the total number of results.
        IncludeTotalCount = true
    };

    // Add fields to include in the search results.
    options.Select.Add("HotelName");
    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);

    // This variable communicates the total number of pages to the view.
    model.pageCount = ((int)model.resultList.TotalCount + GlobalVariables.ResultsPerPage - 1) /
GlobalVariables.ResultsPerPage;

    // This variable communicates the page number being displayed to the view.
    model.currentPage = page;

    // Calculate the range of page numbers to display.
    if (page == 0)
    {
        leftMostPage = 0;
    }
    else if (page <= leftMostPage)
    {
        // Trigger a switch to a lower page range.
        leftMostPage = Math.Max(page - GlobalVariables.PageRangeDelta, 0);
    }
    else if (page >= leftMostPage + GlobalVariables.MaxPageRange - 1)
    {
        // Trigger a switch to a higher page range.
        leftMostPage = Math.Min(page - GlobalVariables.PageRangeDelta, model.pageCount -
GlobalVariables.MaxPageRange);
    }
    model.leftMostPage = leftMostPage;

    // Calculate the number of page numbers to display.
    model.pageRange = Math.Min(model.pageCount - leftMostPage, GlobalVariables.MaxPageRange);

    return View("Index", model);
}

```

- Finally, make a small change to the view. The variable **resultList.Results.TotalCount** will now contain the number of results returned in one page (3 in our example), not the total number. Because we set the **IncludeTotalCount** to true, the variable **resultList.TotalCount** now contains the total number of results. So locate where the number of results is displayed in the view, and change it to the following code.

```
// Show the result count.  
<p class="sampleText">  
    @Model.resultList.TotalCount Results  
</p>  
  
var results = Model.resultList.GetResults().ToList();  
  
@for (var i = 0; i < results.Count; i++)  
{  
    // Display the hotel name and description.  
    @Html.TextAreaFor(m => results[i].Document.HotelName, new { @class = "box1" })  
    @Html.TextArea($"desc{i}", results[i].Document.Description, new { @class = "box2" })  
}
```

NOTE

There is a minor performance hit when setting `IncludeTotalCount` to true, as this total needs to be calculated by Azure Cognitive Search. With complex data sets, there is a warning that the value returned is an *approximation*. Because the hotel search corpus is small, it will be accurate.

Compile and run the app

Now select **Start Without Debugging** (or press the F5 key).

1. Search on a string that returns plenty of results (such as "wifi"). Can you page neatly through the results?



Hotels Search - Paging



19 Results

Super Deluxe Inn & Suites

Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor/Outdoor Pool, Fitness Center, Florida Green certified, Starbucks Coffee, HDTV

Double Sanctuary Resort

5* Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Conde Nast Traveler. Free WiFi, Flexible check in/out, Fitness Center & Nespresso in room.

Veteran Right Track

Free Shuttle to the airport and casinos. Free breakfast and WiFi.

|< < 1 2 3 4 5 > >|

2. Try clicking on the right-most, and later, left-most page numbers. Do the page numbers adjust appropriately to center the page you are on?
3. Are the "first" and "last" options useful? Some commercial search engines use these options, and others do not.
4. Go to the last page of results. The last page is the only page that may contain less than **ResultsPerPage** results.



Hotels Search - Paging

wifi



19 Results

Whitefish Lodge & Suites

Located on in the heart of the forest. Enjoy Warm Weather, Beach Club Services, Natural Hot Springs, Airport Shuttle.

|< < 3 4 5 6 7 > >|

5. Type in "town", and click search. No paging options are displayed if the results are fewer than one page.



Hotels Search - Paging

town



2 Results

Twin Dome Motel

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.

Commander Hotel

Right in the heart of campus. From meetings in town or gameday, enjoy our prime location between the union and steps away from the stadium.

Save this project and continue to the next section for an alternative form of paging.

Extend your app with infinite scrolling

Infinite scrolling is triggered when a user scrolls a vertical scroll bar to the last of the results being displayed. In this event, a call to the search service is made for the next page of results. If there are no more results, nothing is returned and the vertical scroll bar does not change. If there are more results, they are appended to the current page, and the scroll bar changes to show that more results are available.

An important point to note is that the current page is not replaced, but rather extended to show the additional results. A user can always scroll back up to the first results of the search.

To implement infinite scrolling, let's start with the project before any of the page number scrolling elements were added. On GitHub, this is the [FirstAzureSearchApp](#) solution.

Add paging fields to the model

1. First, add a **paging** property to the **SearchData** class (in the `SearchData.cs` model file).

```
// Record if the next page is requested.  
public string paging { get; set; }
```

This variable is a string, which holds "next" if the next page of results should be sent, or be null for the first page of a search.

2. In the same file, and within the namespace, add a global variable class with one property. In MVC, global variables are declared in their own static class. **ResultsPerPage** sets the number of results per page.

```
public static class GlobalVariables  
{  
    public static int ResultsPerPage  
    {  
        get  
        {  
            return 3;  
        }  
    }  
}
```

Add a vertical scroll bar to the view

1. Locate the section of the `index.cshtml` file that displays the results (it starts with the `@if (Model != null)`).
2. Replace the section with the code below. The new `<div>` section is around the area that should be scrollable, and adds both an `overflow-y` attribute and a call to an `onscroll` function called "scrolled()", like so.

```
@if (Model != null)  
{  
    // Show the result count.  
    <p class="sampleText">  
        @Model.resultList.TotalCount Results  
    </p>  
  
    var results = Model.resultList.GetResults().ToList();  
  
    <div id="myDiv" style="width: 800px; height: 450px; overflow-y: scroll;" onscroll="scrolled()">  
  
        <!-- Show the hotel data. -->  
        @for (var i = 0; i < results.Count; i++)  
        {  
            // Display the hotel name and description.  
            @Html.TextAreaFor(m => results[i].Document.HotelName, new { @class = "box1" })  
            @Html.TextArea($"desc{i}", results[i].Document.Description, new { @class = "box2" })  
        }  
    </div>
```

3. Directly underneath the loop, after the `</div>` tag, add the `scrolled` function.

```

<script>
    function scrolled() {
        if (myDiv.offsetHeight + myDiv.scrollTop >= myDiv.scrollHeight) {
            $.getJSON("/Home/NextAsync", function (data) {
                var div = document.getElementById('myDiv');

                // Append the returned data to the current list of hotels.
                for (var i = 0; i < data.length; i += 2) {
                    div.innerHTML += '\n<textarea class="box1">' + data[i] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box2">' + data[i + 1] + '</textarea>';
                }
            });
        }
    }
</script>

```

The **if** statement in the script above tests whether the user has scrolled to the bottom of the vertical scroll bar. If they have, a call to the **Home** controller is made to an action called **NextAsync**. No other information is needed by the controller, it will return the next page of data. This data is then formatted using identical HTML styles as the original page. If no results are returned, nothing is appended and things stay as they are.

Handle the Next action

There are only three actions that need to be sent to the controller: the first running of the app, which calls **Index()**, the first search by the user, which calls **Index(model)**, and then the subsequent calls for more results via **Next(model)**.

1. Open the home controller file and delete the **RunQueryAsync** method from the original tutorial.
2. Replace the **Index(model)** action with the following code. It now handles the **paging** field when it is null, or set to "next", and handles the call to Azure Cognitive Search.

```

public async Task<ActionResult> Index(SearchData model)
{
    try
    {
        InitSearch();

        int page;

        if (model.paging != null && model.paging == "next")
        {
            // Increment the page.
            page = (int)TempData["page"] + 1;

            // Recover the search text.
            model.searchText = TempData["searchfor"].ToString();
        }
        else
        {
            // First call. Check for valid text input.
            if (model.searchText == null)
            {
                model.searchText = "";
            }
            page = 0;
        }

        // Setup the search parameters.
        var options = new SearchOptions
        {
            SearchMode = SearchMode.All,

            // Skip past results that have already been returned.
            Skip = page * GlobalVariables.ResultsPerPage,

            // Take only the next page worth of results.
            Size = GlobalVariables.ResultsPerPage,

            // Include the total number of results.
            IncludeTotalCount = true
        };

        // Specify which fields to include in results.
        options.Select.Add("HotelName");
        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);

        // Ensure TempData is stored for the next call.
        TempData["page"] = page;
        TempData["searchfor"] = model.searchText;
    }
    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "1" });
    }

    return View("Index", model);
}

```

Similar to the numbered paging method, we use the **Skip** and **Size** search settings to request just the data we need is returned.

3. Add the **NextAsync** action to the home controller. Notice how it returns a list, each hotel adding two

elements to the list: a hotel name and a hotel description. This format is set to match the `scrolled` function's use of the returned data in the view.

```
public async Task<ActionResult> NextAsync(SearchData model)
{
    // Set the next page setting, and call the Index(model) action.
    model.paging = "next";
    await Index(model).ConfigureAwait(false);

    // Create an empty list.
    var nextHotels = new List<string>();

    // Add a hotel name, then description, to the list.
    await foreach (var searchResult in model.resultList.GetResultsAsync())
    {
        nextHotels.Add(searchResult.Document.HotelName);
        nextHotels.Add(searchResult.Document.Description);
    }

    // Rather than return a view, return the list of data.
    return new JsonResult(nextHotels);
}
```

4. If you get a syntax error on `List<string>`, add the following `using` directive to the head of the controller file.

```
using System.Collections.Generic;
```

Compile and run your project

Now select **Start Without Debugging** (or press the F5 key).

1. Enter a term that will give plenty of results (such as "pool") and then test the vertical scroll bar. Does it trigger a new page of results?



Hotels Search - Infinite Scroll

pool



16 Results

Twin Dome Motel

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.

Gacc Capital

Chic hotel near the city. High-rise hotel in downtown, walking distance to theaters, restaurants and shops, complete with wellness programs.

Pull'r Inn Motel

The hotel rooms and suites offer the perfect blend of beauty and elegance. Our rooms will elevate your stay, whether you're traveling for business, celebrating a honeymoon, or just looking for a remarkable getaway. With views of the valley or the iconic fountains right from your suite, your stay will be nothing short of unforgettable.

TIP

To ensure that a scroll bar appears on the first page, the first page of results must slightly exceed the height of the area they are being displayed in. In our example .box1 has a height of 30 pixels, .box2 has a height of 100 pixels and a bottom margin of 24 pixels. So each entry uses 154 pixels. Three entries will take up $3 \times 154 = 462$ pixels. To ensure that a vertical scroll bar appears, a height to the display area must be set that is smaller than 462 pixels, even 461 works. This issue only occurs on the first page, after that a scroll bar is sure to appear. The line to update is: `<div id="myDiv" style="width: 800px; height: 450px; overflow-y: scroll;" onscroll="scrolled()>`.

2. Scroll down all the way to the bottom of the results. Notice how all information is now on the one view page. You can scroll all the way back to the top without triggering any server calls.

More sophisticated infinite scrolling systems might use the mouse wheel, or similar other mechanism, to trigger the loading of a new page of results. We will not be taking infinite scrolling any further in these tutorials, but it has a certain charm to it as it avoids extra mouse clicks, and you might want to investigate other options further.

Takeaways

Consider the following takeaways from this project:

- Numbered paging is useful for searches where the order of the results is somewhat arbitrary, meaning there may well be something of interest to your users on the later pages.
- Infinite scrolling is useful when the order of results is particularly important. For example, if the results are ordered on the distance from the center of a destination city.
- Numbered paging allows for better navigation. For example, a user can remember that an interesting result was on page 6, whereas no such easy reference exists in infinite scrolling.
- Infinite scrolling has an easy appeal, scrolling up and down with no page numbers to click on.

- A key feature of infinite scrolling is that results are appended to an existing page, not replacing that page, which is efficient.
- Temporary storage persists for only one call, and needs to be reset to survive additional calls.

Next steps

Paging is fundamental to a search experience. With paging well covered, the next step is to improve the user experience further by adding type-ahead searches.

[C# Tutorial: Add autocompletion and suggestions - Azure Cognitive Search](#)

Tutorial: Add autocomplete and suggestions using the .NET SDK

10/4/2020 • 12 minutes to read • [Edit Online](#)

Learn how to implement autocomplete (typeahead queries and suggested results) when a user starts typing into a search box. In this tutorial, we will show autocompleted queries and suggested results separately, and then together. A user may only have to type two or three characters to locate all the results that are available.

In this tutorial, you learn how to:

- Add suggestions
- Add highlighting to the suggestions
- Add autocomplete
- Combine autocompletion and suggestions

Overview

This tutorial adds autocompletion and suggested results to the previous [Add paging to search results](#) tutorial.

A finished version of the code in this tutorial can be found in the following project:

- [3-add-typeahead \(GitHub\)](#)

Prerequisites

- [2a-add-paging \(GitHub\)](#) solution. This project can either be your own version built from the previous tutorial or a copy from GitHub.

This tutorial has been updated to use the [Azure.Search.Documents \(version 11\)](#) package. For an earlier version of the .NET SDK, see [Microsoft.Azure.Search \(version 10\)](#) code sample.

Add suggestions

Let's start with the simplest case of offering up alternatives to the user: a drop-down list of suggested results.

1. In the index.cshtml file, change `@id` of the `TextBoxFor` statement to `azureautosuggest`.

```
@Html.TextBoxFor(m => m.searchText, new { @class = "searchBox", @id = "azureautosuggest" }) <input  
value="" class="searchBoxSubmit" type="submit">
```

2. Following this statement, after the closing `</div>`, enter this script. This script leverages the [Autocomplete widget](#) from the open-source jQuery UI library to present the dropdown list of suggested results.

```

<script>
    $("#azureautosuggest").autocomplete({
        source: "/Home/SuggestAsync?highlights=false&fuzzy=false",
        minLength: 2,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        }
    });
</script>

```

The ID "azureautosuggest" connects the above script to the search box. The source option of the widget is set to a Suggest method that calls the Suggest API with two query parameters: **highlights** and **fuzzy**, both set to false in this instance. Also, a minimum of two characters is needed to trigger the search.

Add references to jQuery scripts to the view

1. To access the jQuery library, change the <head> section of the view file to the following code:

```

<head>
    <meta charset="utf-8">
    <title>Typeahead</title>
    <link href="https://code.jquery.com/ui/1.12.1/themes/start/jquery-ui.css"
          rel="stylesheet">
    <script src="https://code.jquery.com/jquery-1.10.2.js"></script>
    <script src="https://code.jquery.com/ui/1.12.1/jquery-ui.js"></script>

    <link rel="stylesheet" href="~/css/hotels.css" />
</head>

```

2. Because we are introducing a new jQuery reference, we also need to remove, or comment out, the default jQuery reference in the _Layout.cshtml file (in the **Views/Shared** folder). Locate the following lines, and comment out the first script line as shown. This change avoids clashing references to jQuery.

```

<environment include="Development">
    <!-- <script src="~/lib/jquery/dist/jquery.js"></script> -->
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>

```

Now we can use the predefined Autocomplete jQuery functions.

Add the Suggest action to the controller

1. In the home controller, add the **SuggestAsync** action (after the **PageAsync** action).

```

public async Task<ActionResult> SuggestAsync(bool highlights, bool fuzzy, string term)
{
    InitSearch();

    // Setup the suggest parameters.
    var options = new SuggestOptions()
    {
        UseFuzzyMatching = fuzzy,
        Size = 8,
    };

    if (highlights)
    {
        options.HighlightPreTag = "<b>";
        options.HighlightPostTag = "</b>";
    }

    // Only one suggester can be specified per index. It is defined in the index schema.
    // The name of the suggester is set when the suggester is specified by other API calls.
    // The suggester for the hotel database is called "sg", and simply searches the hotel name.
    var suggestResult = await _searchClient.SuggestAsync<Hotel>(term, "sg",
options).ConfigureAwait(false);

    // Convert the suggested 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 **Size** parameter specifies how many results to return (if unspecified, the default is 5). A *suggester* is specified on the search index when the index is created. In the sample hotels index hosted by Microsoft, the suggester name is "sg", and it searches for suggested matches exclusively in the **HotelName** field.

Fuzzy matching allows "near misses" to be included in the output, up to one edit distance. If the **highlights** parameter is set to true, then bold HTML tags are added to the output. We will set both parameters to true in the next section.

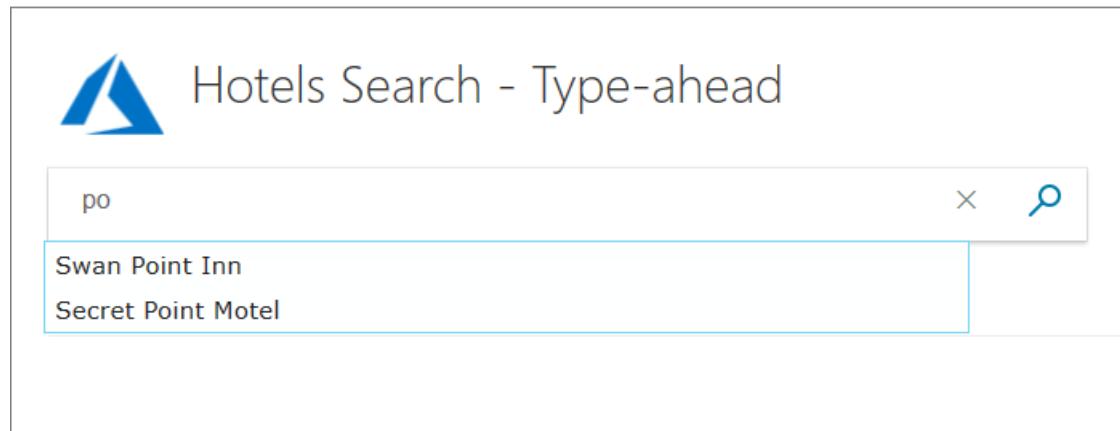
2. You may get some syntax errors. If so, add the following two **using** statements to the top of the file.

```

using System.Collections.Generic;
using System.Linq;

```

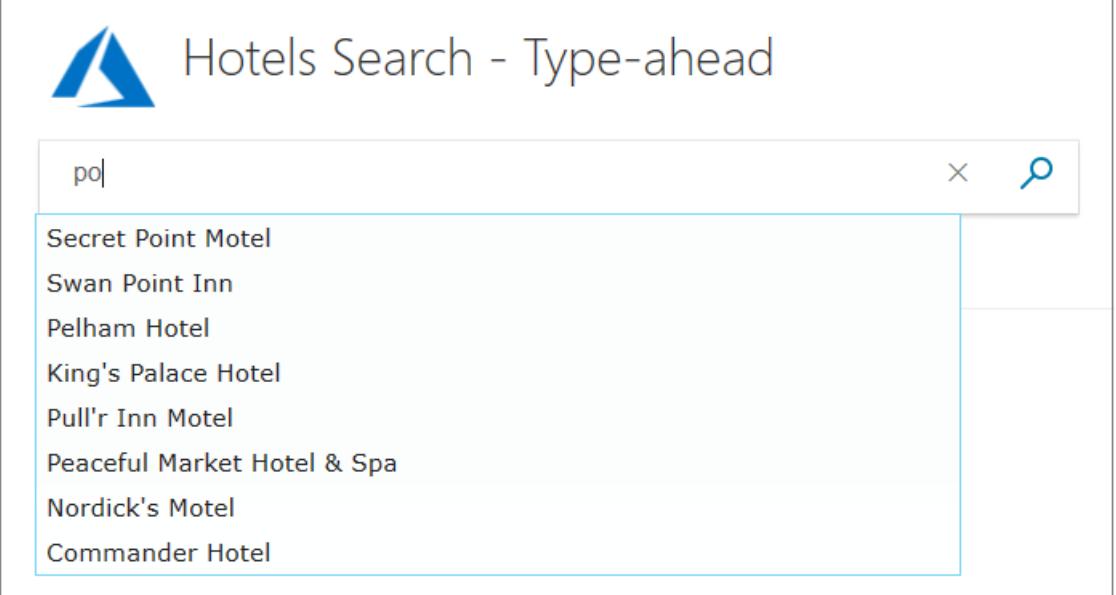
3. Run the app. Do you get a range of options when you enter "po", for example? Now try "pa".



Notice that the letters you enter *must* start a word, and not simply be included within the word.

4. In the view script, set **&fuzzy** to true, and run the app again. Now enter "po". Notice that the search

assumes you got one letter wrong.



If you are interested, the [Lucene query syntax in Azure Cognitive Search](#) describes the logic used in fuzzy searches in detail.

Add highlighting to the suggestions

We can improve the appearance of the suggestions to the user by setting the **highlights** parameter to true. However, first we need to add some code to the view to display the bolded text.

1. In the view (index.cshtml), add the following script after the `"azureautosuggest"` script described previously.

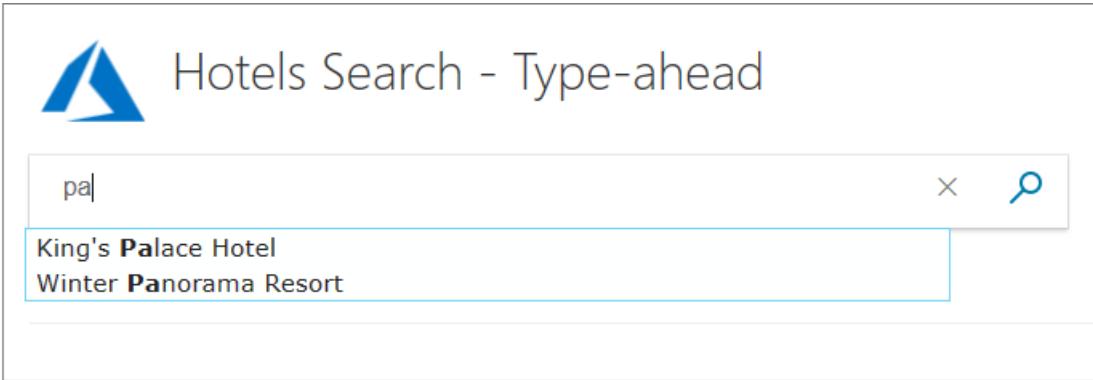
```
<script>
    var updateTextbox = function (event, ui) {
        var result = ui.item.value.replace(/<\/?[^>]+(>|$/g, "");
        $("#azuresuggesthighlights").val(result);
        return false;
    };

    $("#azuresuggesthighlights").autocomplete({
        html: true,
        source: "/home/suggest?highlights=true&fuzzy=false&",
        minLength: 2,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        },
        select: updateTextbox,
        focus: updateTextbox
    }).data("ui-autocomplete")._renderItem = function (ul, item) {
        return $(<li></li>")
            .data("item.autocomplete", item)
            .append("<a>" + item.label + "</a>")
            .appendTo(ul);
    };
</script>
```

2. Now change the ID of the text box so it reads as follows.

```
@Html.TextBoxFor(m => m.searchText, new { @class = "searchBox", @id = "azuresuggesthighlights" })
<input value="" class="searchBoxSubmit" type="submit">
```

- Run the app again, and you should see your entered text bolded in the suggestions. Try typing "pa".



The logic used in the highlighting script above is not foolproof. If you enter a term that appears twice in the same name, the bolded results are not quite what you would want. Try typing "mo".

One of the questions a developer needs to answer is, when is a script working "well enough", and when should its quirks be addressed. We will not be taking highlighting any further in this tutorial, but finding a precise algorithm is something to consider if highlighting is not effective for your data. For more information, see [Hit highlighting](#).

Add autocomplete

Another variation, slightly different from suggestions, is autocomplete (sometimes called "type-ahead") that completes a query term. Again, we will start with the simplest implementation, before improving the user experience.

- Enter the following script into the view, following your previous scripts.

```
<script>
    $("#azureautocompletebasic").autocomplete({
        source: "/Home/Autocomplete",
        minLength: 2,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        }
    });
</script>
```

- Now change the ID of the text box, so it reads as follows.

```
@Html.TextBoxFor(m => m.searchText, new { @class = "searchBox", @id = "azureautocompletebasic" })
<input value="" class="searchBoxSubmit" type="submit">
```

- In the home controller, enter the **AutocompleteAsync** action after the **SuggestAsync** action.

```

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();

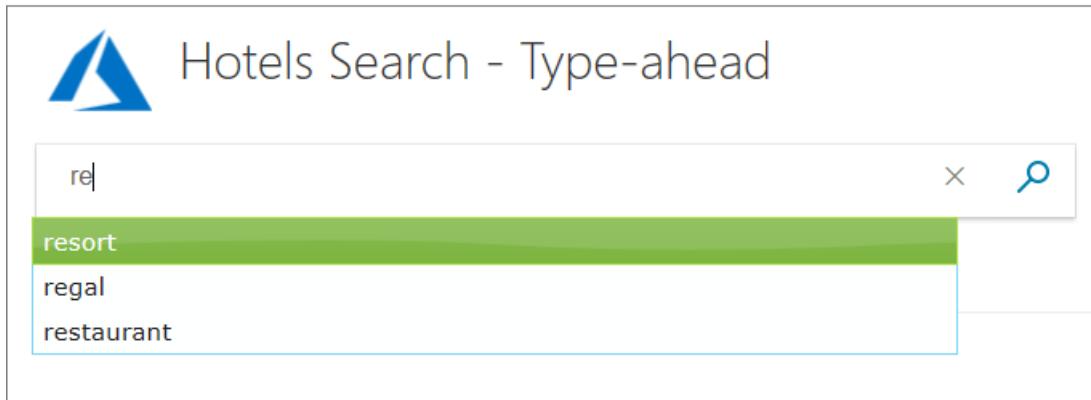
    return new JsonResult(autocomplete);
}

```

Notice that we are using the same *suggester* function, called "sg", in the autocomplete search as we did for suggestions (so we are only trying to autocomplete the hotel names).

There are a range of **AutocompleteMode** settings, and we are using **OneTermWithContext**. Refer to [Autocomplete API](#) for a description of additional options.

- Run the app. Notice how the range of options displayed in the drop-down list are single words. Try typing words starting with "re". Notice how the number of options reduces as more letters are typed.



As it stands, the suggestions script you ran earlier is probably more helpful than this autocompletion script. To make autocompletion more user-friendly, consider using it with suggested results.

Combine autocompletion and suggestions

Combining autocompletion and suggestions is the most complex of our options, and probably provides the best user experience. What we want is to display, inline with the text that is being typed, is the first choice of Azure Cognitive Search for autocompleting the text. Also, we want a range of suggestions as a drop-down list.

There are libraries that offer this functionality - often called "inline autocompletion" or a similar name. However, we are going to natively implement this feature so that you can explore the APIs. We are going to start work on the controller first in this example.

- Add an action to the controller that returns just one autocompletion result, along with a specified number of suggestions. We will call this action **AutoCompleteAndSuggestAsync**. In the home controller, add the following action, following your other new actions.

```

public async Task<ActionResult> AutoCompleteAndSuggestAsync(string term)
{
    InitSearch();

    // Setup the type-ahead search parameters.
    var ap = new AutocompleteOptions()
    {
        Mode = AutocompleteMode.OneTermWithContext,
        Size = 1,
    };
    var autocompleteResult = await _searchClient.AutocompleteAsync(term, "sg", ap);

    // Setup the suggest search parameters.
    var sp = new SuggestOptions()
    {
        Size = 8,
    };

    // Only one suggester can be specified per index. The name of the suggester is set when the
    suggester is specified by other API calls.
    // The suggester for the hotel database is called "sg" and simply searches the hotel name.
    var suggestResult = await _searchClient.SuggestAsync<Hotel>(term, "sg", sp).ConfigureAwait(false);

    // Create an empty list.
    var results = new List<string>();

    if (autocompleteResult.Value.Results.Count > 0)
    {
        // Add the top result for type-ahead.
        results.Add(autocompleteResult.Value.Results[0].Text);
    }
    else
    {
        // There were no type-ahead suggestions, so add an empty string.
        results.Add("");
    }

    for (int n = 0; n < suggestResult.Value.Results.Count; n++)
    {
        // Now add the suggestions.
        results.Add(suggestResult.Value.Results[n].Text);
    }

    // Return the list.
    return new JsonResult(results);
}

```

One autocompletion option is returned at the top of the **results** list, followed by all the suggestions.

2. In the view, first we implement a trick so that a light gray autocompletion word is rendered right under bolder text being entered by the user. HTML includes relative positioning for this purpose. Change the **TextBoxFor** statement (and its surrounding **<div>** statements) to the following, noting that a second search box identified as **underneath** is right under our normal search box, by pulling this search box 39 pixels off of its default location!

```

<div id="underneath" class="searchBox" style="position: relative; left: 0; top: 0">
</div>

<div id="searchinput" class="searchBoxForm" style="position: relative; left: 0; top: -39px">
    @Html.TextBoxFor(m => m.searchText, new { @class = "searchBox", @id = "azureautocomplete" }) <input
    value="" class="searchBoxSubmit" type="submit">
</div>

```

Notice we are changing the ID again, to **azureautocomplete** in this case.

- Also in the view, enter the following script, after all the scripts you have entered so far. The script is lengthy and complex due to the variety of input behaviors that it handles.

```
<script>
    $('#azureautocomplete').autocomplete({
        delay: 500,
        minLength: 2,
        position: {
            my: "left top",
            at: "left-23 bottom+10"
        },

        // Use Ajax to set up a "success" function.
        source: function (request, response) {
            var controllerUrl = "/Home/AutoCompleteAndSuggestAsync?term=" +
                $('#azureautocomplete').val();
            $.ajax({
                url: controllerUrl,
                dataType: "json",
                success: function (data) {
                    if (data && data.length > 0) {

                        // Show the autocomplete suggestion.
                        document.getElementById("underneath").innerHTML = data[0];

                        // Remove the top suggestion as it is used for inline autocomplete.
                        var array = new Array();
                        for (var n = 1; n < data.length; n++) {
                            array[n - 1] = data[n];
                        }

                        // Show the drop-down list of suggestions.
                        response(array);
                    } else {

                        // Nothing is returned, so clear the autocomplete suggestion.
                        document.getElementById("underneath").innerHTML = "";
                    }
                }
            });
        }
    });

    // Complete on TAB.
    // Clear on ESC.
    // Clear if backspace to less than 2 characters.
    // Clear if any arrow key hit as user is navigating the suggestions.
    $('#azureautocomplete').keydown(function (evt) {

        var suggestedText = document.getElementById("underneath").innerHTML;
        if (evt.keyCode === 9 /* TAB */ && suggestedText.length > 0) {
            $('#azureautocomplete').val(suggestedText);
            return false;
        } else if (evt.keyCode === 27 /* ESC */) {
            document.getElementById("underneath").innerHTML = "";
            $('#azureautocomplete').val("");
        } else if (evt.keyCode === 8 /* Backspace */) {
            if ($('#azureautocomplete').val().length < 2) {
                document.getElementById("underneath").innerHTML = "";
            }
        } else if (evt.keyCode >= 37 && evt.keyCode <= 40 /* Any arrow key */) {
            document.getElementById("underneath").innerHTML = "";
        }
    });
});
```

```

// Character replace function.
function setCharAt(str, index, chr) {
    if (index > str.length - 1) return str;
    return str.substr(0, index) + chr + str.substr(index + 1);
}

// This function is needed to clear the "underneath" text when the user clicks on a suggestion, and
to
// correct the case of the autocomplete option when it does not match the case of the user input.
// The interval function is activated with the input, blur, change, or focus events.
$("#azureautocomplete").on("input blur change focus", function (e) {

    // Set a 2 second interval duration.
    var intervalDuration = 2000,
        interval = setInterval(function () {

            // Compare the autocorrect suggestion with the actual typed string.
            var inputText = document.getElementById("azureautocomplete").value;
            var autoText = document.getElementById("underneath").innerHTML;

            // If the typed string is longer than the suggestion, then clear the suggestion.
            if (inputText.length > autoText.length) {
                document.getElementById("underneath").innerHTML = "";
            } else {

                // If the strings match, change the case of the suggestion to match the case of the
                typed input.
                if (autoText.toLowerCase().startsWith(inputText.toLowerCase())) {
                    for (var n = 0; n < inputText.length; n++) {
                        autoText = setCharAt(autoText, n, inputText[n]);
                    }
                    document.getElementById("underneath").innerHTML = autoText;
                } else {
                    // The strings do not match, so clear the suggestion.
                    document.getElementById("underneath").innerHTML = "";
                }
            }

            // If the element loses focus, stop the interval checking.
            if (!$input.is(':focus')) clearInterval(interval);

        }, intervalDuration);
    });

```

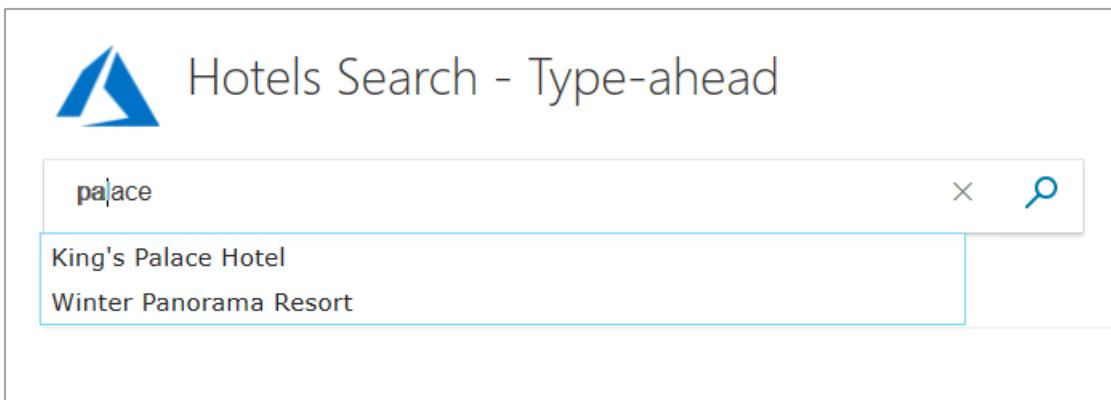
Notice how the `interval` function is used to both clear the underlying text when it no longer matches what the user is typing, and also to set the same case (upper or lower) as the user is typing (as "pa" matches "PA", "pA", "Pa" when searching), so that the overlaid text is neat.

Read through the comments in the script to get a fuller understanding.

- Finally, we need to make a minor adjustment to two HTML class to make them transparent. Add the following line to the `searchBoxForm` and `searchBox` classes, in the `hotels.css` file.

```
background: rgba(0,0,0,0);
```

- Now run the app. Enter "pa" into the search box. Do you get "palace" as the autocomplete suggestion, along with two hotels that contain "pa"?



6. Try tabbing to accept the autocomplete suggestion, and try selecting suggestions using the arrow keys and tab key, and try again using the mouse and a single click. Verify that the script handles all these situations neatly.

You may decide that it is simpler to load in a library that offers this feature for you, but now you know at least one way to get inline autocompletion to work.

Takeaways

Consider the following takeaways from this project:

- Autocompletion (also known as "type-ahead") and suggestions can enable the user to type only a few keys to locate exactly what they want.
- Autocompletion and suggestions working together can provide a rich user experience.
- Always test autocompletion functions with all forms of input.
- Using the `setInterval` function can be useful in verifying and correcting UI elements.

Next steps

In the next tutorial, we have a look at another way of improving the user experience, using facets to narrow searches with a single click.

[C# Tutorial: Use facets to aid navigation - Azure Cognitive Search](#)

Tutorial: Add faceted navigation using the .NET SDK

10/4/2020 • 10 minutes to read • [Edit Online](#)

Facets enable self-directed navigation by providing a set of links for filtering results. In this tutorial, a faceted navigation structure is placed on the left side of the page, with labels and clickable text to trim the results.

In this tutorial, you learn how to:

- Set model properties as *IsFacetable*
- Add facet navigation to your app

Overview

Facets are based on fields in your search index. A query request that includes facet=[string] provides the field to facet by. It's common to include multiple facets, such as `&facet=category&facet=amenities`, each one separated by an ampersand (&) character. Implementing a faceted navigation structure requires that you specify both facets and filters. The filter is used on a click event to narrow results. For example, clicking "budget" filters the results based on that criteria.

This tutorial extends the paging project created in the [Add paging to search results](#) tutorial.

A finished version of the code in this tutorial can be found in the following project:

- [4-add-facet-navigation \(GitHub\)](#)

Prerequisites

- [2a-add-paging \(GitHub\)](#) solution. This project can either be your own version built from the previous tutorial or a copy from GitHub.

This tutorial has been updated to use the [Azure.Search.Documents \(version 11\)](#) package. For an earlier version of the .NET SDK, see [Microsoft.Azure.Search \(version 10\)](#) code sample.

Set model properties as *IsFacetable*

In order for a model property to be located in a facet search, it must be tagged with **IsFacetable**.

1. Examine the **Hotel** class. **Category** and **Tags**, for example, are tagged as **IsFacetable**, but **HotelName** and **Description** are not.

```

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 Room[] Rooms { get; set; }
}

```

2. We will not be changing any tags as part of this tutorial, so close the hotel.cs file unaltered.

NOTE

A facet search will throw an error if a field requested in the search is not tagged appropriately.

Add facet navigation to your app

For this example, we are going to enable the user to select one category of hotel, or one amenity, from lists of links shown to the left of the results. The user starts by entering some search text, then progressively narrow the results of the search by selecting a category or amenity.

It's the controller's job to pass the lists of facets to the view. To maintain the user selections as the search progresses, we use temporary storage as the mechanism for preserving state.



wifi

**Category:**

Budget (5)
Luxury (5)
Resort and Spa (5)
Boutique (3)
Extended-Stay (1)

Amenities:

free wifi (11)
laundry service (6)
24-hour front desk service (5)
pool (5)
restaurant (5)
concierge (4)
continental breakfast (4)
free parking (4)
view (4)
coffee in lobby (3)
air conditioning (2)
bar (1)

19 Results

Super Deluxe Inn & Suites

Complimentary Airport Shuttle & WiFi. Book Now and save - Spacious All Suite Hotel, Indoor/Outdoor Pool, Fitness Center, Florida Green certified, Starbucks Coffee, HDTV

Category: Boutique

Amenities: bar, free wifi, free wifi

Double Sanctuary Resort

5* Luxury Hotel - Biggest Rooms in the city. #1 Hotel in the area listed by Conde Nast Traveler. Free WiFi, Flexible check in/out, Fitness Center & Nespresso in room.

Category: Resort and Spa

Amenities: view, laundry service, free wifi

Pull'r Inn Motel

The hotel rooms and suites offer the perfect blend of beauty and elegance. Our rooms will elevate your stay, whether you're traveling for business, celebrating a honeymoon, or just looking for a remarkable getaway. With views of the valley or the iconic fountains right from your suite, your stay will be nothing short of unforgettable.

|< < 1 2 3 4 5 > >|

Add filter strings to the SearchData model

1. Open the `SearchData.cs` file, and add string properties to the `SearchData` class, to hold the facet filter strings.

```
public string categoryFilter { get; set; }  
public string amenityFilter { get; set; }
```

Add the Facet action method

The home controller needs one new action, `Facet`, and updates to its existing `Index` and `Page` actions, and to the `RunQueryAsync` method.

1. Replace the `Index(SearchData model)` action method.

```
public async Task<ActionResult> Index(SearchData model)
{
    try
    {
        // Ensure the search string is valid.
        if (model.searchText == null)
        {
            model.searchText = "";
        }

        // Make the search call for the first page.
        await RunQueryAsync(model, 0, 0, "", "").ConfigureAwait(false);
    }
    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "1" });
    }

    return View(model);
}
```

2. Replace the **PageAsync(SearchData model)** action method.

```

public async Task<ActionResult> PageAsync(SearchData model)
{
    try
    {
        int page;

        // Calculate the page that should be displayed.
        switch (model.paging)
        {
            case "prev":
                page = (int)TempData["page"] - 1;
                break;

            case "next":
                page = (int)TempData["page"] + 1;
                break;

            default:
                page = int.Parse(model.paging);
                break;
        }

        // Recover the leftMostPage.
        int leftMostPage = (int)TempData["leftMostPage"];

        // Recover the filters.
        string catFilter = TempData["categoryFilter"].ToString();
        string ameFilter = TempData["amenityFilter"].ToString();

        // Recover the search text.
        model.searchText = TempData["searchfor"].ToString();

        // Search for the new page.
        await RunQueryAsync(model, page, leftMostPage, catFilter, ameFilter);
    }

    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "2" });
    }
    return View("Index", model);
}

```

3. Add a **FacetAsync(SearchData model)** action method, to be activated when the user clicks on a facet link.
The model will contain either a category or amenity search filter. Add it after the **PageAsync** action.

```

public async Task<ActionResult> FacetAsync(SearchData model)
{
    try
    {
        // Filters set by the model override those stored in temporary data.
        string catFilter;
        string ameFilter;
        if (model.categoryFilter != null)
        {
            catFilter = model.categoryFilter;
        } else
        {
            catFilter = TempData["categoryFilter"].ToString();
        }

        if (model.amenityFilter != null)
        {
            ameFilter = model.amenityFilter;
        } else
        {
            ameFilter = TempData["amenityFilter"].ToString();
        }

        // Recover the search text.
        model.searchText = TempData["searchfor"].ToString();

        // Initiate a new search.
        await RunQueryAsync(model, 0, 0, catFilter, ameFilter).ConfigureAwait(false);
    }
    catch
    {
        return View("Error", new ErrorViewModel { RequestId = "2" });
    }

    return View("Index", model);
}

```

Set up the search filter

When a user selects a certain facet, for example, they click on the **Resort and Spa** category, then only hotels that are specified as this category should be returned in the results. To narrow a search in this way, we need to set up a *filter*.

1. Replace the **RunQueryAsync** method with the following code. Primarily, it takes a category filter string, and an amenity filter string, and sets the **Filter** parameter of the **SearchOptions**.

```

private async Task<ActionResult> RunQueryAsync(SearchData model, int page, int leftMostPage, string
catFilter, string ameFilter)
{
    InitSearch();

    string facetFilter = "";

    if (catFilter.Length > 0 && ameFilter.Length > 0)
    {
        // Both facets apply.
        facetFilter = $"{catFilter} and {ameFilter}";
    } else
    {
        // One, or zero, facets apply.
        facetFilter = $"{catFilter}{ameFilter}";
    }

    var options = new SearchOptions
    {
        ...
    }

```

```

        Filter = facetFilter,
        SearchMode = SearchMode.All,
        // Skip past results that have already been returned.
        Skip = page * GlobalVariables.ResultsPerPage,
        // Take only the next page worth of results.
        Size = GlobalVariables.ResultsPerPage,
        // Include the total number of results.
        IncludeTotalCount = true,
    };

    // Return information on the text, and number, of facets in the data.
    options.Facets.Add("Category,count:20");
    options.Facets.Add("Tags,count:20");

    // Enter Hotel property names into this list, so only these values will be returned.
    options.Select.Add("HotelName");
    options.Select.Add("Description");
    options.Select.Add("Category");
    options.Select.Add("Tags");

    // 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);

    // This variable communicates the total number of pages to the view.
    model.pageCount = ((int)model.resultList.TotalCount + GlobalVariables.ResultsPerPage - 1) /
GlobalVariables.ResultsPerPage;

    // This variable communicates the page number being displayed to the view.
    model.currentPage = page;

    // Calculate the range of page numbers to display.
    if (page == 0)
    {
        leftMostPage = 0;
    }
    else if (page <= leftMostPage)
    {
        // Trigger a switch to a lower page range.
        leftMostPage = Math.Max(page - GlobalVariables.PageRangeDelta, 0);
    }
    else if (page >= leftMostPage + GlobalVariables.MaxPageRange - 1)
    {
        // Trigger a switch to a higher page range.
        leftMostPage = Math.Min(page - GlobalVariables.PageRangeDelta, model.pageCount -
GlobalVariables.MaxPageRange);
    }
    model.leftMostPage = leftMostPage;

    // Calculate the number of page numbers to display.
    model.pageRange = Math.Min(model.pageCount - leftMostPage, GlobalVariables.MaxPageRange);

    // Ensure Temp data is stored for the next call.
    TempData["page"] = page;
    TempData["leftMostPage"] = model.leftMostPage;
    TempData["searchfor"] = model.searchText;
    TempData["categoryFilter"] = catFilter;
    TempData["amenityFilter"] = ameFilter;

    // Return the new view.
    return View("Index", model);
}

```

Notice that the **Category** and **Tags** properties are added to the list of **Select** items to return. This addition

is not a requirement for facet navigation to work, but we use this information to verify the filters are working correctly.

Add lists of facet links to the view

The view is going to require some significant changes.

1. Start by opening the hotels.css file (in the wwwroot/css folder), and add the following classes.

```
.facetlist {
    list-style: none;
}

.facetchecks {
    width: 250px;
    display: normal;
    color: #666;
    margin: 10px;
    padding: 5px;
}

.facetheader {
    font-size: 10pt;
    font-weight: bold;
    color: darkgreen;
}
```

2. For the view, organize the output into a table, to neatly align the facet lists on the left, and the results on the right. Open the index.cshtml file. Replace the entire contents of the HTML <body> tags, with the following code.

```
<body>
    @using (Html.BeginForm("Index", "Home", FormMethod.Post))
    {
        <table>
            <tr>
                <td></td>
                <td>
                    <h1 class="sampleTitle">
                        
                        Hotels Search - Facet Navigation
                    </h1>
                </td>
            </tr>

            <tr>
                <td></td>
                <td>
                    <!-- Display the search text box, with the search icon to the right of it.-->
                    <div class="searchBoxForm">
                        @Html.TextBoxFor(m => m.searchText, new { @class = "searchBox" }) <input
value="" class="searchBoxSubmit" type="submit">
                    </div>
                </td>
            </tr>

            <tr>
                <td valign="top">
                    <div id="facetplace" class="facetchecks">
                        @if (Model != null && Model.resultList != null)
                        {
                            List<string> categories = Model.resultList.Facets["Category"].Select(x =>
x.Value.ToString()).ToList();
                        }
                    </div>
                </td>
            </tr>
        </table>
    }
</body>
```

```

        if (categories.Count > 0)
        {
            <h5 class="facetheader">Category:</h5>
            <ul class="facetlist">
                @for (var c = 0; c < categories.Count; c++)
                {
                    var facetLink = $"{categories[c]}"
({Model.resultList.Facets["Category"]}[c].Count}");
                    <li>
                        @Html.ActionLink(facetLink, "FacetAsync", "Home", new {
categoryFilter = $"Category eq '{categories[c]}'" }, null)
                    </li>
                }
            </ul>
        }

        List<string> tags = Model.resultList.Facets["Tags"].Select(x =>
x.Value.ToString()).ToList();

        if (tags.Count > 0)
        {
            <h5 class="facetheader">Amenities:</h5>
            <ul class="facetlist">
                @for (var c = 0; c < tags.Count; c++)
                {
                    var facetLink = $"{tags[c]} ({Model.resultList.Facets["Tags"]
[c].Count})";
                    <li>
                        @Html.ActionLink(facetLink, "FacetAsync", "Home", new {
amenityFilter = $"Tags/any(t: t eq '{tags[c]}')" }, null)
                    </li>
                }
            </ul>
        }
    </div>
</td>
<td valign="top">
    <div id="resultsplace">
        @if (Model != null && Model.resultList != null)
        {
            // Show the result count.
            <p class="sampleText">
                @Model.resultList.TotalCount Results
            </p>

            var results = Model.resultList.GetResults().ToList();

            @for (var i = 0; i < results.Count; i++)
            {
                string amenities = string.Join(", ", results[i].Document.Tags);

                string fullDescription = results[i].Document.Description;
                fullDescription += $"{"\nCategory: {results[i].Document.Category}}";
                fullDescription += $"{"\nAmenities: {amenities}}";

                // Display the hotel name and description.
                @Html.TextAreaFor(m => results[i].Document.HotelName, new { @class =
"box1" })
                @Html.TextArea($"desc{i}", fullDescription, new { @class = "box2" })
            }
        }
    </div>
</td>
</tr>
<tr>
    <td></td>

```

```

<td valign="top">
    @if (Model != null && Model.pageCount > 1)
    {
        // If there is more than one page of results, show the paging buttons.
        <table>
            <tr>
                <td class="tdPage">
                    @if (Model.currentPage > 0)
                    {
                        <p class="pageButton">
                            @Html.ActionLink("|<", "PageAsync", "Home", new { paging =
                            "0" }, null)
                        </p>
                    }
                    else
                    {
                        <p class="pageButtonDisabled">|</p>
                    }
                </td>

                <td class="tdPage">
                    @if (Model.currentPage > 0)
                    {
                        <p class="pageButton">
                            @Html.ActionLink("<", "PageAsync", "Home", new { paging =
                            "prev" }, null)
                        </p>
                    }
                    else
                    {
                        <p class="pageButtonDisabled">&lt;</p>
                    }
                </td>

                @for (var pn = Model.leftMostPage; pn < Model.leftMostPage +
                    Model.pageRange; pn++)
                {
                    <td class="tdPage">
                        @if (Model.currentPage == pn)
                        {
                            // Convert displayed page numbers to 1-based and not 0-
                            based.
                            <p class="pageSelected">@(pn + 1)</p>
                        }
                        else
                        {
                            <p class="pageButton">
                                @Html.ActionLink((pn + 1).ToString(), "PageAsync",
                                "Home", new { paging = @pn }, null)
                            </p>
                        }
                    </td>
                }

                <td class="tdPage">
                    @if (Model.currentPage < Model.pageCount - 1)
                    {
                        <p class="pageButton">
                            @Html.ActionLink(">", "PageAsync", "Home", new { paging =
                            "next" }, null)
                        </p>
                    }
                    else
                    {
                        <p class="pageButtonDisabled">&gt;</p>
                    }
                </td>

                <td class="tdPage">

```

```

        @if (Model.currentPage < Model.pageCount - 1)
        {
            <p class="pageButton">
                @Html.ActionLink(">|", "PageAsync", "Home", new { paging =
Model.pageCount - 1 }, null)
            </p>
        }
        else
        {
            <p class="pageButtonDisabled">&gt;|</p>
        }
    </td>
</tr>
</table>
}
</td>
</tr>
</table>
}
</body>

```

Notice the use of the `Html.ActionLink` call. This call communicates valid filter strings to the controller, when the user clicks a facet link.

Run and test the app

The advantage of facet navigation to the user is that they can narrow searches with a single click, which we can show in the following sequence.

1. Run the app, type "airport" as the search text. Verify that the list of facets appears neatly to the left. These facets are all that apply to hotels that have "airport" in their text data, with a count of how often they occur.

Hotels Search - Facet Navigation

airport

Category:

- Luxury (4)
- Resort and Spa (4)
- Boutique (1)

Amenities:

- continental breakfast (5)
- 24-hour front desk service (4)
- free parking (3)
- free wifi (3)
- view (2)
- air conditioning (1)
- bar (1)
- coffee in lobby (1)
- concierge (1)
- laundry service (1)
- pool (1)
- restaurant (1)

Veteran Right Track
Free Shuttle to the airport and casinos. Free breakfast and WiFi.
Category: Resort and Spa
Amenities: 24-hour front desk service, restaurant, concierge

Nova Hotel & Spa
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
Amenities: pool, continental breakfast, free parking

Suites At Bellevue Square
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
Amenities: continental breakfast, air conditioning, 24-hour front desk service

< < 1 2 3 > >|

2. Click the **Resort and Spa** category. Verify all results are in this category.



Hotels Search - Facet Navigation

**Category:**[Resort and Spa \(4\)](#)**Amenities:**

- [24-hour front desk service \(2\)](#)
- [continental breakfast \(2\)](#)
- [air conditioning \(1\)](#)
- [coffee in lobby \(1\)](#)
- [concierge \(1\)](#)
- [free parking \(1\)](#)
- [laundry service \(1\)](#)
- [pool \(1\)](#)
- [restaurant \(1\)](#)
- [view \(1\)](#)

4 Results

Veteran Right Track

Free Shuttle to the airport and casinos. Free breakfast and WiFi.

Category: Resort and Spa

Amenities: 24-hour front desk service, restaurant, concierge

Nova Hotel & Spa

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

Amenities: pool, continental breakfast, free parking

Suites At Bellevue Square

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

Amenities: continental breakfast, air conditioning, 24-hour front desk service

|< < 1 2 > >|

3. Click the **continental breakfast** amenity. Verify all results are still in the "Resort and Spa" category, with the selected amenity.



Hotels Search - Facet Navigation

**Category:**[Resort and Spa \(2\)](#)**Amenities:**

- [continental breakfast \(2\)](#)
- [24-hour front desk service \(1\)](#)
- [air conditioning \(1\)](#)
- [free parking \(1\)](#)
- [pool \(1\)](#)

2 Results

Nova Hotel & Spa

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

Amenities: pool, continental breakfast, free parking

Suites At Bellevue Square

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

Amenities: continental breakfast, air conditioning, 24-hour front desk service

4. Try selecting any other category, then one amenity, and view the narrowing results. Then try the other way around, one amenity, then one category. Send an empty search to reset the page.

NOTE

When one selection is made in a facet list (such as category) it will override any previous selection within the category list.

Takeaways

Consider the following takeaways from this project:

- It is imperative to mark each facetable field with the **IsFacetable** property for inclusion in facet navigation.
- Facets are combined with filters to reduce the results.
- Facets are cumulative, with each selection building on the previous one to further narrow results.

Next steps

In the next tutorial, we look at ordering results. Up to this point, results are ordered simply in the order that they are located in the database.

[C# tutorial: Order the results- Azure Cognitive Search](#)

Tutorial: Order search results using the .NET SDK

10/4/2020 • 22 minutes to read • [Edit Online](#)

Throughout this tutorial series, results have been returned and displayed in a [default order](#). In this tutorial, you will add primary and secondary sort criteria. As an alternative to ordering based on numerical values, the final example shows how to rank results based on a custom scoring profile. We will also go a bit deeper into the display of *complex types*.

In this tutorial, you learn how to:

- Order results based on one property
- Order results based on multiple properties
- Filter results based on a distance from a geographical point
- Order results based on a scoring profile

Overview

This tutorial extends the infinite scrolling project created in the [Add paging to search results](#) tutorial.

A finished version of the code in this tutorial can be found in the following project:

- [5-order-results \(GitHub\)](#)

Prerequisites

- [2b-add-infinite-scroll \(GitHub\)](#) solution. This project can either be your own version built from the previous tutorial or a copy from GitHub.

This tutorial has been updated to use the [Azure.Search.Documents \(version 11\)](#) package. For an earlier version of the .NET SDK, see [Microsoft.Azure.Search \(version 10\)](#) code sample.

Order results based on one property

When ordering results based on one property, such as hotel rating, we not only want the ordered results, we also want confirmation that the order is correct. Adding the rating field to the results allows us to confirm the results are sorted correctly.

In this exercise, we will also add a bit more to the display of results: the cheapest room rate, and the most expensive room rate, for each hotel.

There is no need to modify any of the models to enable ordering. Only the view and the controller require updates. Start by opening the home controller.

Add the OrderBy property to the search parameters

1. Add the `OrderBy` option to the name of the property. In the `Index(SearchData model)` method, add the following line to the search parameters.

```
OrderBy = new[] { "Rating desc" },
```

NOTE

The default order is ascending, though you can add **asc** to the property to make this clear. Descending order is specified by adding **desc**.

2. Now run the app, and enter any common search term. The results may or may not be in the correct order, as neither you as the developer, nor the user, has any easy way of verifying the results!
3. Let's make it clear the results are ordered on rating. First, replace the **box1** and **box2** classes in the `hotels.css` file with the following classes (these classes are all the new ones we need for this tutorial).

```

textarea.box1A {
    width: 324px;
    height: 32px;
    border: none;
    background-color: azure;
    font-size: 14pt;
    color: blue;
    padding-left: 5px;
    text-align: left;
}

textarea.box1B {
    width: 324px;
    height: 32px;
    border: none;
    background-color: azure;
    font-size: 14pt;
    color: blue;
    text-align: right;
    padding-right: 5px;
}

textarea.box2A {
    width: 324px;
    height: 32px;
    border: none;
    background-color: azure;
    font-size: 12pt;
    color: blue;
    padding-left: 5px;
    text-align: left;
}

textarea.box2B {
    width: 324px;
    height: 32px;
    border: none;
    background-color: azure;
    font-size: 12pt;
    color: blue;
    text-align: right;
    padding-right: 5px;
}

textarea.box3 {
    width: 648px;
    height: 100px;
    border: none;
    background-color: azure;
    font-size: 12pt;
    padding-left: 5px;
    margin-bottom: 24px;
}

```

TIP

Browsers usually cache css files, and this can lead to an old css file being used, and your edits ignored. A good way round this is to add a query string with a version parameter to the link. For example:

```
<link rel="stylesheet" href="~/css/hotels.css?v1.1" />
```

Update the version number if you think an old css file is being used by your browser.

4. Add the **Rating** property to the **Select** parameter, in the **Index(SearchData model)** method.

```
Select = new[] { "HotelName", "Description", "Rating"},
```

5. Open the view (index.cshtml) and replace the rendering loop (`<!-- Show the hotel data. -->`) with the following code.

```
<!-- Show the hotel data. -->
@for (var i = 0; i < Model.resultList.Results.Count; i++)
{
    var ratingText = $"Rating: {Model.resultList.Results[i].Document.Rating}";

    // Display the hotel details.
    @Html.TextArea($"name{i}", Model.resultList.Results[i].Document.HotelName, new { @class = "box1A" })
    @Html.TextArea($"rating{i}", ratingText, new { @class = "box1B" })
    @Html.TextArea($"desc{i}", Model.resultList.Results[i].Document.Description, new {
@class = "box3" })
}
```

6. The rating needs to be available both in the first displayed page, and in the subsequent pages that are called via the infinite scroll. For the latter of these two situations, we need to update both the **Next** action in the controller, and the **scrolled** function in the view. Starting with the controller, change the **Next** method to the following code. This code creates and communicates the rating text.

```
public async Task<ActionResult> Next(SearchData model)
{
    // Set the next page setting, and call the Index(model) action.
    model.paging = "next";
    await Index(model);

    // Create an empty list.
    var nextHotels = new List<string>();

    // Add a hotel details to the list.
    for (int n = 0; n < model.resultList.Results.Count; n++)
    {
        var ratingText = $"Rating: {model.resultList.Results[n].Document.Rating}";

        // Add three strings to the list.
        nextHotels.Add(model.resultList.Results[n].Document.HotelName);
        nextHotels.Add(ratingText);
        nextHotels.Add(model.resultList.Results[n].Document.Description);
    }

    // Rather than return a view, return the list of data.
    return new JsonResult(nextHotels);
}
```

7. Now update the **scrolled** function in the view, to display the rating text.

```

<script>
    function scrolled() {
        if (myDiv.offsetHeight + myDiv.scrollTop >= myDiv.scrollHeight) {
            $.getJSON("/Home/Next", function (data) {
                var div = document.getElementById('myDiv');

                // Append the returned data to the current list of hotels.
                for (var i = 0; i < data.length; i += 3) {
                    div.innerHTML += '\n<textarea class="box1A">' + data[i] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box1B">' + data[i + 1] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box3">' + data[i + 2] + '</textarea>';
                }
            });
        }
    }
</script>

```

8. Now run the app again. Search on any common term, such as "wifi", and verify that the results are ordered by descending order of hotel rating.

wifi

19 Results

Summer Wind Resort	Rating: 4.9
Steps from the Convention Center. Located in the heart of downtown with modern rooms with stunning city views, 24-7 dining options, free WiFi and easy valet parking	
Pull'r Inn Motel	Rating: 4.7
The hotel rooms and suites offer the perfect blend of beauty and elegance. Our rooms will elevate your stay, whether you're traveling for business, celebrating a honeymoon, or just looking for a remarkable getaway. With views of the valley or the iconic fountains right from your suite, your stay will be nothing short of unforgettable.	
Motteler's Thunderbird Motel	Rating: 4.4
Book Now & Save. Clean, Comfortable rooms at the lowest price. Enjoy complimentary coffee and tea in common areas.	

You will notice that several hotels have an identical rating, and so their appearance in the display is again the order in which the data is found, which is arbitrary.

Before we look into adding a second level of ordering, let's add some code to display the range of room rates. We are adding this code to both show extracting data from a *complex type*, and also so we can discuss ordering results based on price (cheapest first perhaps).

Add the range of room rates to the view

1. Add properties containing the cheapest and most expensive room rate to the Hotel.cs model.

```
// Room rate range
public double cheapest { get; set; }
public double expensive { get; set; }
```

2. Calculate the room rates at the end of the **Index(SearchData model)** action, in the home controller. Add the calculations after the storing of temporary data.

```
// Ensure TempData is stored for the next call.
TempData["page"] = page;
TempData["searchfor"] = model.searchText;

// Calculate the room rate ranges.
for (int n = 0; n < model.resultList.Results.Count; n++)
{
    // Calculate room rates.
    var cheapest = 0d;
    var expensive = 0d;

    for (var r = 0; r < model.resultList.Results[n].Document.Rooms.Length; r++)
    {
        var rate = model.resultList.Results[n].Document.Rooms[r].BaseRate;
        if (rate < cheapest || cheapest == 0)
        {
            cheapest = (double)rate;
        }
        if (rate > expensive)
        {
            expensive = (double)rate;
        }
    }
    model.resultList.Results[n].Document.cheapest = cheapest;
    model.resultList.Results[n].Document.expensive = expensive;
}
```

3. Add the **Rooms** property to the **Select** parameter in the **Index(SearchData model)** action method of the controller.

```
Select = new[] { "HotelName", "Description", "Rating", "Rooms" },
```

4. Change the rendering loop in the view to display the rate range for the first page of results.

```
<!-- Show the hotel data. -->
@for (var i = 0; i < Model.resultList.Results.Count; i++)
{
    var rateText = $"Rates from ${Model.resultList.Results[i].Document.cheapest} to
${Model.resultList.Results[i].Document.expensive}";
    var ratingText = $"Rating: {Model.resultList.Results[i].Document.Rating}";

    // Display the hotel details.
    @Html.TextArea($"name{i}", Model.resultList.Results[i].Document.HotelName, new { @class
= "box1A" })
    @Html.TextArea($"rating{i}", ratingText, new { @class = "box1B" })
    @Html.TextArea($"rates{i}" , rateText, new { @class = "box2A" })
    @Html.TextArea($"desc{i}" , Model.resultList.Results[i].Document.Description, new {
@class = "box3" })
}
```

5. Change the **Next** method in the home controller to communicate the rate range, for subsequent pages of results.

```

public async Task<ActionResult> Next(SearchData model)
{
    // Set the next page setting, and call the Index(model) action.
    model.paging = "next";
    await Index(model);

    // Create an empty list.
    var nextHotels = new List<string>();

    // Add a hotel details to the list.
    for (int n = 0; n < model.resultList.Results.Count; n++)
    {
        var ratingText = $"Rating: {model.resultList.Results[n].Document.Rating}";
        var rateText = $"Rates from ${model.resultList.Results[n].Document.cheapest} to
${model.resultList.Results[n].Document.expensive}";

        // Add strings to the list.
        nextHotels.Add(model.resultList.Results[n].Document.HotelName);
        nextHotels.Add(ratingText);
        nextHotels.Add(rateText);
        nextHotels.Add(model.resultList.Results[n].Document.Description);
    }

    // Rather than return a view, return the list of data.
    return new JsonResult(nextHotels);
}

```

6. Update the **scrolled** function in the view, to handle the room rates text.

```

<script>
    function scrolled() {
        if (myDiv.offsetHeight + myDiv.scrollTop >= myDiv.scrollHeight) {
            $.getJSON("/Home/Next", function (data) {
                var div = document.getElementById('myDiv');

                // Append the returned data to the current list of hotels.
                for (var i = 0; i < data.length; i += 4) {
                    div.innerHTML += '\n<textarea class="box1A">' + data[i] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box1B">' + data[i + 1] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box2A">' + data[i + 2] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box3">' + data[i + 4] + '</textarea>';
                }
            });
        }
    }
</script>

```

7. Run the app, and verify the room rate ranges are displayed.



Hotels Search - Ordering Results

pool



16 Results

Pull'r Inn Motel

Rating: 4.7

Rates from \$64.99 to \$268.99

The hotel rooms and suites offer the perfect blend of beauty and elegance. Our rooms will elevate your stay, whether you're traveling for business, celebrating a honeymoon, or just looking for a remarkable getaway. With views of the valley or the iconic fountains right from your suite, your stay will be nothing short of unforgettable.

Travel Resort

Rating: 4.2

Rates from \$72.99 to \$259.99

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.

Days Hotel

Rating: 4.2

Rates from \$60.99 to \$266.99

The **OrderBy** property of the search parameters will not accept an entry such as **Rooms.BaseRate** to provide the cheapest room rate, even if the rooms were already sorted on rate. In this case, the rooms are not sorted on rate. In order to display hotels in the sample data set, ordered on room rate, you would have to sort the results in your home controller, and send these results to the view in the desired order.

Order results based on multiple values

The question now is how to differentiate between hotels with the same rating. One good way would be to order on the basis of the last time the hotel was renovated. In other words, the more recently the hotel was renovated, the higher the hotel appears in the results.

1. To add a second level of ordering, change the **OrderBy** and **Select** properties in the **Index(SearchData model)** method to include the **LastRenovationDate** property.

```
OrderBy = new[] { "Rating desc", "LastRenovationDate desc" },
Select = new[] { "HotelName", "Description", "Rating", "Rooms", "LastRenovationDate" },
```

TIP

Any number of properties can be entered in the **OrderBy** list. If hotels had the same rating and renovation date, a third property could be entered to differentiate between them.

2. Again, we need to see the renovation date in the view, just to be certain the ordering is correct. For such a thing as a renovation, probably just the year is required. Change the rendering loop in the view to the following code.

```

<!-- Show the hotel data. -->
@for (var i = 0; i < Model.resultList.Results.Count; i++)
{
    var rateText = $"Rates from ${Model.resultList.Results[i].Document.cheapest} to
${Model.resultList.Results[i].Document.expensive}";
    var lastRenovatedText = $"Last renovated: {
Model.resultList.Results[i].Document.LastRenovationDate.Value.Year}";
    var ratingText = $"Rating: ${Model.resultList.Results[i].Document.Rating}";

    // Display the hotel details.
    @Html.TextArea($"name{i}", Model.resultList.Results[i].Document.HotelName, new { @class
= "box1A" })
        @Html.TextArea($"rating{i}", ratingText, new { @class = "box1B" })
        @Html.TextArea($"rates{i}" , rateText, new { @class = "box2A" })
        @Html.TextArea($"renovation{i}" , lastRenovatedText, new { @class = "box2B" })
        @Html.TextArea($"desc{i}" , Model.resultList.Results[i].Document.Description, new {
@class = "box3" })
}

```

3. Change the **Next** method in the home controller, to forward the year component of the last renovation date.

```

public async Task<ActionResult> Next(SearchData model)
{
    // Set the next page setting, and call the Index(model) action.
    model.paging = "next";
    await Index(model);

    // Create an empty list.
    var nextHotels = new List<string>();

    // Add a hotel details to the list.
    for (int n = 0; n < model.resultList.Results.Count; n++)
    {
        var ratingText = $"Rating: ${model.resultList.Results[n].Document.Rating}";
        var rateText = $"Rates from ${model.resultList.Results[n].Document.cheapest} to
${model.resultList.Results[n].Document.expensive}";
        var lastRenovatedText = $"Last renovated:
{model.resultList.Results[n].Document.LastRenovationDate.Value.Year}";

        // Add strings to the list.
        nextHotels.Add(model.resultList.Results[n].Document.HotelName);
        nextHotels.Add(ratingText);
        nextHotels.Add(rateText);
        nextHotels.Add(lastRenovatedText);
        nextHotels.Add(model.resultList.Results[n].Document.Description);
    }

    // Rather than return a view, return the list of data.
    return new JsonResult(nextHotels);
}

```

4. Change the **scrolled** function in the view to display the renovation text.

```

<script>
    function scrolled() {
        if (myDiv.offsetHeight + myDiv.scrollTop >= myDiv.scrollHeight) {
            $.getJSON("/Home/Next", function (data) {
                var div = document.getElementById('myDiv');

                // Append the returned data to the current list of hotels.
                for (var i = 0; i < data.length; i += 5) {
                    div.innerHTML += '\n<textarea class="box1A">' + data[i] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box1B">' + data[i + 1] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box2A">' + data[i + 2] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box2B">' + data[i + 3] + '</textarea>';
                    div.innerHTML += '\n<textarea class="box3">' + data[i + 4] + '</textarea>';
                }
            });
        }
    }
</script>

```

- Run the app. Search on a common term, such as "pool" or "view", and verify that hotels with the same rating are now displayed in descending order of renovation date.

The screenshot shows a web application interface for searching hotels. At the top, there is a logo consisting of three blue triangles pointing upwards, followed by the text "Hotels Search - Ordering Results". Below this is a search bar containing the word "pool" and a magnifying glass icon. Underneath the search bar, it says "16 Results". The results are listed in a grid format:

Hotel Name	Rating	Renovation Date
King's Palace Hotel	Rating: 3.5	Last renovated: 2005
Gacc Capital	Rating: 3.5	Last renovated: 2000
Pelham Hotel	Rating: 3.5	Last renovated: 1983

Each result row contains the hotel name, its rating, and the year it was last renovated. The renovation date is listed in descending order from most recent to oldest. The interface has a light blue background and a vertical scrollbar on the right side of the results area.

Filter results based on a distance from a geographical point

Rating, and renovation date, are examples of properties that are best displayed in a descending order. An alphabetical listing would be an example of a good use of ascending order (for example, if there was just one **OrderBy** property, and it was set to **HotelName** then an alphabetical order would be displayed). However, for our sample data, distance from a geographical point would be more appropriate.

To display results based on geographical distance, several steps are required.

1. Filter out all hotels that are outside of a specified radius from the given point, by entering a filter with longitude, latitude, and radius parameters. Longitude is given first to the POINT function. Radius is in kilometers.

```
// "Location" must match the field name in the Hotel class.  
// Distance (the radius) is in kilometers.  
// Point order is Longitude then Latitude.  
Filter = $"geo.distance(Location, geography'POINT({model.lon} {model.lat})') le {model.radius}",
```

2. The above filter does *not* order the results based on distance, it just removes the outliers. To order the results, enter an **OrderBy** setting that specifies the geoDistance method.

```
OrderBy = new[] { $"geo.distance(Location, geography'POINT({model.lon} {model.lat})') asc" },
```

3. Although the results were returned by Azure Cognitive Search using a distance filter, the calculated distance between the data and the specified point is *not* returned. Recalculate this value in the view, or controller, if you want to display it in the results.

The following code will calculate the distance between two lat/lon points.

```
const double EarthRadius = 6371;  
  
public static double Degrees2Radians(double deg)  
{  
    return deg * Math.PI / 180;  
}  
  
public static double DistanceInKm( double lat1,  double lon1, double lat2, double lon2)  
{  
    double dlon = Degrees2Radians(lon2 - lon1);  
    double dlat = Degrees2Radians(lat2 - lat1);  
  
    double a = (Math.Sin(dlat / 2) * Math.Sin(dlat / 2)) + Math.Cos(Degrees2Radians(lat1)) *  
    Math.Cos(Degrees2Radians(lat2)) * (Math.Sin(dlon / 2) * Math.Sin(dlon / 2));  
    double angle = 2 * Math.Atan2(Math.Sqrt(a), Math.Sqrt(1 - a));  
    return angle * EarthRadius;  
}
```

4. Now you have to tie these concepts together. However, these code snippets are as far as our tutorial goes, building a map-based app is left as an exercise for the reader. To take this example further, consider either entering a city name with a radius, or locating a point on a map, and selecting a radius. To investigate these options further, see the following resources:

- [Azure Maps Documentation](#)
- [Find an address using the Azure Maps search service](#)

Order results based on a scoring profile

The examples given in the tutorial so far show how to order on numerical values (rating, renovation date, geographical distance), providing an *exact* process of ordering. However, some searches and some data do not lend themselves to such an easy comparison between two data elements. Azure Cognitive Search includes the concept of *scoring*. *Scoring profiles* can be specified for a set of data that can be used to provide more complex and qualitative comparisons, that should be most valuable when, say, comparing text-based data to decide which should be displayed first.

Scoring profiles are not defined by users, but typically by administrators of a data set. Several scoring profiles have been set up on the hotels data. Let's look at how a scoring profile is defined, then try writing code to search on

them.

How scoring profiles are defined

Let's look at three examples of scoring profiles, and consider how each *should* affect the results order. As an app developer, you do not write these profiles, they are written by the data administrator, however, it is helpful to look at the syntax.

1. This is the default scoring profile for the hotels data set, used when you do not specify any **OrderBy** or **ScoringProfile** parameter. This profile boosts the *score* for a hotel if the search text is present in the hotel name, description, or list of tags (amenities). Notice how the weights of the scoring favor certain fields. If the search text appears in another field, not listed below, it will have a weight of 1. Obviously, the higher the score, the earlier a result appears in the view.

```
{  
    "name": "boostByField",  
    "text": {  
        "weights": {  
            "HotelName": 2,  
            "Description": 1.5,  
            "Description_fr": 1.5,  
            "Tags": 3  
        }  
    }  
}
```

2. The following scoring profile boosts the score significantly, if a supplied parameter includes one or more of the list of tags (which we are calling "amenities"). The key point of this profile is that a parameter *must* be supplied, containing text. If the parameter is empty, or is not supplied, an error will be thrown.

```
{  
    "name": "boostAmenities",  
    "functions": [  
        {  
            "type": "tag",  
            "fieldName": "Tags",  
            "boost": 5,  
            "tag": {  
                "tagsParameter": "amenities"  
            }  
        }  
    ]  
}
```

3. In this third example, the rating gives a significant boost to the score. The last renovated date will also boost the score, but only if that data falls within 730 days (2 years) of the current date.

```

{
  "name": "renovatedAndHighlyRated",
  "functions": [
    {
      "type": "magnitude",
      "fieldName": "Rating",
      "boost": 20,
      "interpolation": "linear",
      "magnitude": {
        "boostingRangeStart": 0,
        "boostingRangeEnd": 5,
        "constantBoostBeyondRange": false
      }
    },
    {
      "type": "freshness",
      "fieldName": "LastRenovationDate",
      "boost": 10,
      "interpolation": "quadratic",
      "freshness": {
        "boostingDuration": "P730D"
      }
    }
  ]
}

```

Now, let us see if these profiles work as we think they should!

Add code to the view to compare profiles

1. Open the index.cshtml file, and replace the <body> section with the following code.

```

<body>

@using (Html.BeginForm("Index", "Home", FormMethod.Post))
{
  <table>
    <tr>
      <td></td>
      <td>
        <h1 class="sampleTitle">
          
          Hotels Search - Order Results
        </h1>
      </td>
    </tr>
    <tr>
      <td></td>
      <td>
        <!-- Display the search text box, with the search icon to the right of it. -->
        <div class="searchBoxForm">
          @Html.TextBoxFor(m => m.searchText, new { @class = "searchBox" }) <input
          class="searchBoxSubmit" type="submit" value="">
        </div>

        <div class="searchBoxForm">
          <b>&nbsp;Order:&nbsp;</b>
          @Html.RadioButtonFor(m => m.scoring, "Default") Default&nbsp;&nbsp;
          @Html.RadioButtonFor(m => m.scoring, "RatingRenovation") By numerical
          Rating&nbsp;&nbsp;
          @Html.RadioButtonFor(m => m.scoring, "boostAmenities") By Amenities&nbsp;&nbsp;
          @Html.RadioButtonFor(m => m.scoring, "renovatedAndHighlyRated") By Renovated
          date/Rating profile&nbsp;&nbsp;
        </div>
      </td>
    </tr>
  </table>
}

```

```

</tr>

<tr>
    <td valign="top">
        <div id="facetplace" class="facetchecks">

            @if (Model != null && Model.facetText != null)
            {
                <h5 class="facetheader">Amenities:</h5>
                <ul class="facetlist">
                    @for (var c = 0; c < Model.facetText.Length; c++)
                    {
                        <li> @Html.CheckBoxFor(m => m.facetOn[c], new { @id = "check" +
c.ToString() }) @Model.facetText[c] </li>
                    }
                </ul>
            }
        </div>
    </td>
    <td>
        @if (Model != null && Model.resultList != null)
        {
            // Show the total result count.
            <p class="sampleText">
                @Html.DisplayFor(m => m.resultList.Count) Results <br />
            </p>

            <div id="myDiv" style="width: 800px; height: 450px; overflow-y: scroll;" onscroll="scrolled()">

                <!-- Show the hotel data. -->
                @for (var i = 0; i < Model.resultList.Results.Count; i++)
                {
                    var rateText = $"Rates from ${Model.resultList.Results[i].Document.cheapest} to ${Model.resultList.Results[i].Document.expensive}";
                    var lastRenovatedText = $"Last renovated: {Model.resultList.Results[i].Document.LastRenovationDate.Value.Year}";
                    var ratingText = $"Rating: {Model.resultList.Results[i].Document.Rating}";

                    string amenities = string.Join(", ",
Model.resultList.Results[i].Document.Tags);
                    string fullDescription = Model.resultList.Results[i].Document.Description;
                    fullDescription += $"{Environment.NewLine}Amenities: {amenities}";

                    // Display the hotel details.
                    @Html.TextArea($"name{i}", Model.resultList.Results[i].Document.HotelName,
new { @class = "box1A" })
                    @Html.TextArea($"rating{i}", ratingText, new { @class = "box1B" })
                    @Html.TextArea($"rates{i}", rateText, new { @class = "box2A" })
                    @Html.TextArea($"renovation{i}", lastRenovatedText, new { @class = "box2B" })
                }
                @Html.TextArea($"desc{i}", fullDescription, new { @class = "box3" })
            }
        </div>

        <script>
            function scrolled() {
                if (myDiv.offsetHeight + myDiv.scrollTop >= myDiv.scrollHeight) {
                    $.getJSON("/Home/Next", function (data) {
                        var div = document.getElementById('myDiv');

                        // Append the returned data to the current list of hotels.
                        for (var i = 0; i < data.length; i += 5) {
                            div.innerHTML += '\n<textarea class="box1A">' + data[i] +
'</textarea>';
                            div.innerHTML += '<textarea class="box1B">' + data[i + 1] +
'</textarea>';
                            div.innerHTML += '\n<textarea class="box2A">' + data[i + 2] +
'</textarea>';
                        }
                    });
                }
            }
        </script>
    </td>

```

```

'</textarea>';
                    div.innerHTML += '<textarea class="box2B">' + data[i + 3] +
'</textarea>';
                    div.innerHTML += '\n<textarea class="box3">' + data[i + 4] +
'</textarea>';
                }
            });
        }
    
```

2. Open the `SearchData.cs` file, and replace the `SearchData` class with the following code.

```

public class searchData
{
    public searchData()
    {
    }

    // Constructor to initialize the list of facets sent from the controller.
    public searchData(List<string> facets)
    {
        facetText = new string[facets.Count];

        for (int i = 0; i < facets.Count; i++)
        {
            facetText[i] = facets[i];
        }
    }

    // Array to hold the text for each amenity.
    public string[] facetText { get; set; }

    // Array to hold the setting for each amenity.
    public bool[] facetOn { get; set; }

    // The text to search for.
    public string searchText { get; set; }

    // Record if the next page is requested.
    public string paging { get; set; }

    // The list of results.
    public DocumentSearchResult<Hotel> resultlist;

    public string scoring { get; set; }
}

```

3. Open the `hotels.css` file, and add the following HTML classes.

```

.facetlist {
    list-style: none;
}

.facetchecks {
    width: 250px;
    display: normal;
    color: #666;
    margin: 10px;
    padding: 5px;
}

.facetheader {
    font-size: 10pt;
    font-weight: bold;
    color: darkgreen;
}

```

Add code to the controller to specify a scoring profile

1. Open the home controller file. Add the following `using` statement (to aid with creating lists).

```
using System.Linq;
```

2. For this example, we need the initial call to `Index` to do a bit more than just return the initial view. The method now searches for up to 20 amenities to display in the view.

```

public async Task<ActionResult> Index()
{
    InitSearch();

    // Set up the facets call in the search parameters.
    SearchParameters sp = new SearchParameters()
    {
        // Search for up to 20 amenities.
        Facets = new List<string> { "Tags,count:20" },
    };

    DocumentSearchResult<Hotel> searchResult = await _indexClient.Documents.SearchAsync<Hotel>("*",
sp);

    // Convert the results to a list that can be displayed in the client.
    List<string> facets = searchResult.Facets["Tags"].Select(x => x.Value.ToString()).ToList();

    // Initiate a model with a list of facets for the first view.
    searchData model = new searchData(facets);

    // Save the facet text for the next view.
    SaveFacets(model, false);

    // Render the view including the facets.
    return View(model);
}

```

3. We need two private methods to save the facets to temporary storage, and to recover them from temporary storage and populate a model.

```

// Save the facet text to temporary storage, optionally saving the state of the check boxes.
private void SaveFacets(SearchData model, bool saveChecks = false)
{
    for (int i = 0; i < model.facetText.Length; i++)
    {
        TempData["facet" + i.ToString()] = model.facetText[i];
        if (saveChecks)
        {
            TempData["facetOn" + i.ToString()] = model.facetOn[i];
        }
    }
    TempData["facetcount"] = model.facetText.Length;
}

// Recover the facet text to a model, optionally recovering the state of the check boxes.
private void RecoverFacets(SearchData model, bool recoverChecks = false)
{
    // Create arrays of the appropriate length.
    model.facetText = new string[(int)TempData["facetcount"]];
    if (recoverChecks)
    {
        model.facetOn = new bool[(int)TempData["facetcount"]];
    }

    for (int i = 0; i < (int)TempData["facetcount"]; i++)
    {
        model.facetText[i] = TempData["facet" + i.ToString()].ToString();
        if (recoverChecks)
        {
            model.facetOn[i] = (bool)TempData["facetOn" + i.ToString()];
        }
    }
}

```

4. We need to set the **OrderBy** and **ScoringProfile** parameters as necessary. Replace the existing **Index(SearchData model)** method, with the following.

```

public async Task<ActionResult> Index(SearchData model)
{
    try
    {
        InitSearch();

        int page;

        if (model.paging != null && model.paging == "next")
        {
            // Recover the facet text, and the facet check box settings.
            RecoverFacets(model, true);

            // Increment the page.
            page = (int)TempData["page"] + 1;

            // Recover the search text.
            model.searchText = TempData["searchfor"].ToString();
        }
        else
        {
            // First search with text.
            // Recover the facet text, but ignore the check box settings, and use the current model
            settings.
            RecoverFacets(model, false);

            // First call. Check for valid text input, and valid scoring profile.
            if (model.searchText == null)
            {

```

```

        model.searchText = "";
    }
    if (model.scoring == null)
    {
        model.scoring = "Default";
    }
    page = 0;
}

// Set empty defaults for ordering and scoring parameters.
var orderby = new List<string>();
string profile = "";
var scoringParams = new List<ScoringParameter>();

// Set the ordering based on the user's radio button selection.
switch (model.scoring)
{
    case "RatingRenovation":
        orderby.Add("Rating desc");
        orderby.Add("LastRenovationDate desc");
        break;

    case "boostAmenities":
    {
        profile = model.scoring;
        var setAmenities = new List<string>();

        // Create a string list of amenities that have been clicked.
        for (int a = 0; a < model.facetOn.Length; a++)
        {
            if (model.facetOn[a])
            {
                setAmenities.Add(model.facetText[a]);
            }
        }
        if (setAmenities.Count > 0)
        {
            // Only set scoring parameters if there are any.
            var sp = new ScoringParameter("amenities", setAmenities);
            scoringParams.Add(sp);
        }
        else
        {
            // No amenities selected, so set profile back to default.
            profile = "";
        }
    }
    break;

    case "renovatedAndHighlyRated":
        profile = model.scoring;
        break;

    default:
        break;
}

// Setup the search parameters.
var parameters = new SearchParameters
{
    // Set the ordering/scoring parameters.
    OrderBy = orderby,
    ScoringProfile = profile,
    ScoringParameters = scoringParams,

    // Select the data properties to be returned.
    Select = new[] { "HotelName", "Description", "Tags", "Rooms", "Rating",
        "LastRenovationDate" },
    SearchMode = SearchMode.All
}

```

```

SearchMode = SearchMode.All,
Skip = page * GlobalVariables.ResultsPerPage,
Top = GlobalVariables.ResultsPerPage,
IncludeTotalResultCount = true,
};

// For efficiency, the search call should be asynchronous, so use SearchAsync rather than
Search.
model.resultList = await _indexClient.Documents.SearchAsync<Hotel>(model.searchText,
parameters);

// Ensure TempData is stored for the next call.
TempData["page"] = page;
TempData["searchfor"] = model.searchText;
TempData["scoring"] = model.scoring;
SaveFacets(model,true);

// Calculate the room rate ranges.
for (int n = 0; n < model.resultList.Results.Count; n++)
{
    var cheapest = 0d;
    var expensive = 0d;

    for (var r = 0; r < model.resultList.Results[n].Document.Rooms.Length; r++)
    {
        var rate = model.resultList.Results[n].Document.Rooms[r].BaseRate;
        if (rate < cheapest || cheapest == 0)
        {
            cheapest = (double)rate;
        }
        if (rate > expensive)
        {
            expensive = (double)rate;
        }
    }
    model.resultList.Results[n].Document.cheapest = cheapest;
    model.resultList.Results[n].Document.expensive = expensive;
}
}
catch
{
    return View("Error", new ErrorViewModel { RequestId = "1" });
}
return View("Index", model);
}

```

Read through the comments for each of the **switch** selections.

5. We do not need to make any changes to the **Next** action, if you completed the additional code for the previous section on ordering based on multiple properties.

Run and test the app

1. Run the app. You should see a full set of amenities in the view.
2. For ordering, selecting "By numerical Rating" will give you the numerical ordering you have already implemented in this tutorial, with renovation date deciding among hotels of equal rating.

 Hotels Search - Order Results

beach 

Order: Default By numerical Rating By Amenities By Renovated date/Rating profile

Amenities:

- view
- 24-hour front desk service
- laundry service
- air conditioning
- concierge
- pool
- continental breakfast
- free wifi
- restaurant
- coffee in lobby
- free parking
- bar

6 Results

Johnson's Resort	Rating: 4.8
Rates from \$65.99 to \$268.99	Last renovated: 1978
been built with all the comforts of home. Sporting a huge beach with multiple water toys for those sunny summer days and a Lodge full of games for when you just can't swim anymore, there's always something for the family to do. A full marina offers watercraft rentals, boat launch, powered dock slips, canoes (free to use), & fish cleaning facility. Rent pontoon, 14' fishing boats, 16' fishing rigs or jet skis for a fun day or week on the water.	
Lady Of The Lake B & B	Rating: 4.7
Rates from \$60.99 to \$265.99	Last renovated: 1987
Nature is Home on the beach. Save up to 30 percent. Valid Now through the end of the year. Restrictions and blackout may apply. Amenities: laundry service, concierge, view	
Nova Hotel & Spa	Rating: 3.6
Rates from \$60.99 to \$269.99	Last renovated: 1973

3. Now try the "By amenities" profile. Make various selections of amenities, and verify that hotels with those amenities are promoted up the results list.

 Hotels Search - Order Results

beach 

Order: Default By numerical Rating By Amenities By Renovated date/Rating profile

Amenities:

- view
- 24-hour front desk service
- laundry service
- air conditioning
- concierge
- pool
- continental breakfast
- free wifi
- restaurant
- coffee in lobby
- free parking
- bar

6 Results

Nova Hotel & Spa	Rating: 3.6
Rates from \$60.99 to \$269.99	Last renovated: 1973
1 Mile from the airport. Free WiFi, Outdoor Pool, Complimentary Airport Shuttle, 6 miles from the beach & 10 miles from downtown. Amenities: pool, continental breakfast, free parking	
Whitefish Lodge & Suites	Rating: 2.4
Rates from \$92.99 to \$257.99	Last renovated: 1987
Located on in the heart of the forest. Enjoy Warm Weather, Beach Club Services, Natural Hot Springs, Airport Shuttle. Amenities: continental breakfast, free parking, free wifi	
Ocean Air Motel	Rating: 3.5
Rates from \$63.99 to \$254.99	Last renovated: 1951

4. Try the "By Renovated date/Rating profile" to see if you get what you expect. Only recently renovated hotels should get a *freshness* boost.

Resources

For more information, see the following [Add scoring profiles to an Azure Cognitive Search index](#).

Takeaways

Consider the following takeaways from this project:

- Users will expect search results to be ordered, most relevant first.
- Data needs structured so that ordering is easy. We were not able to sort on "cheapest" first easily, as the data is not structured to enable ordering to be done without additional code.
- There can be many levels to ordering, to differentiate between results that have the same value at a higher level of ordering.
- It is natural for some results to be ordered in ascending order (say, distance away from a point), and some in descending order (say, guest's rating).
- Scoring profiles can be defined when numerical comparisons are not available, or not smart enough, for a data set. Scoring each result will help to order and display the results intelligently.

Next steps

You have completed this series of C# tutorials - you should have gained valuable knowledge of the Azure Cognitive Search APIs.

For further reference and tutorials, consider browsing [Microsoft Learn](#), or the other tutorials in the [Azure Cognitive Search Documentation](#).

Tutorial: Index Azure SQL data using the .NET SDK

10/4/2020 • 8 minutes to read • [Edit Online](#)

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.

Home > SQL databases > Create SQL Database

Create SQL Database

Microsoft

[Basics](#) [Networking](#) [Additional settings](#) [Tags](#) [Review + create](#)

Create a SQL database with your preferred configurations. Complete the Basics tab then go to Review + Create to provision with smart defaults, or visit each tab to customize. [Learn more](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * [?](#) [▼](#)

Resource group * [?](#) [▼](#) [Create new](#)

Database details

Enter required settings for this database, including picking a logical server and configuring the compute and storage resources

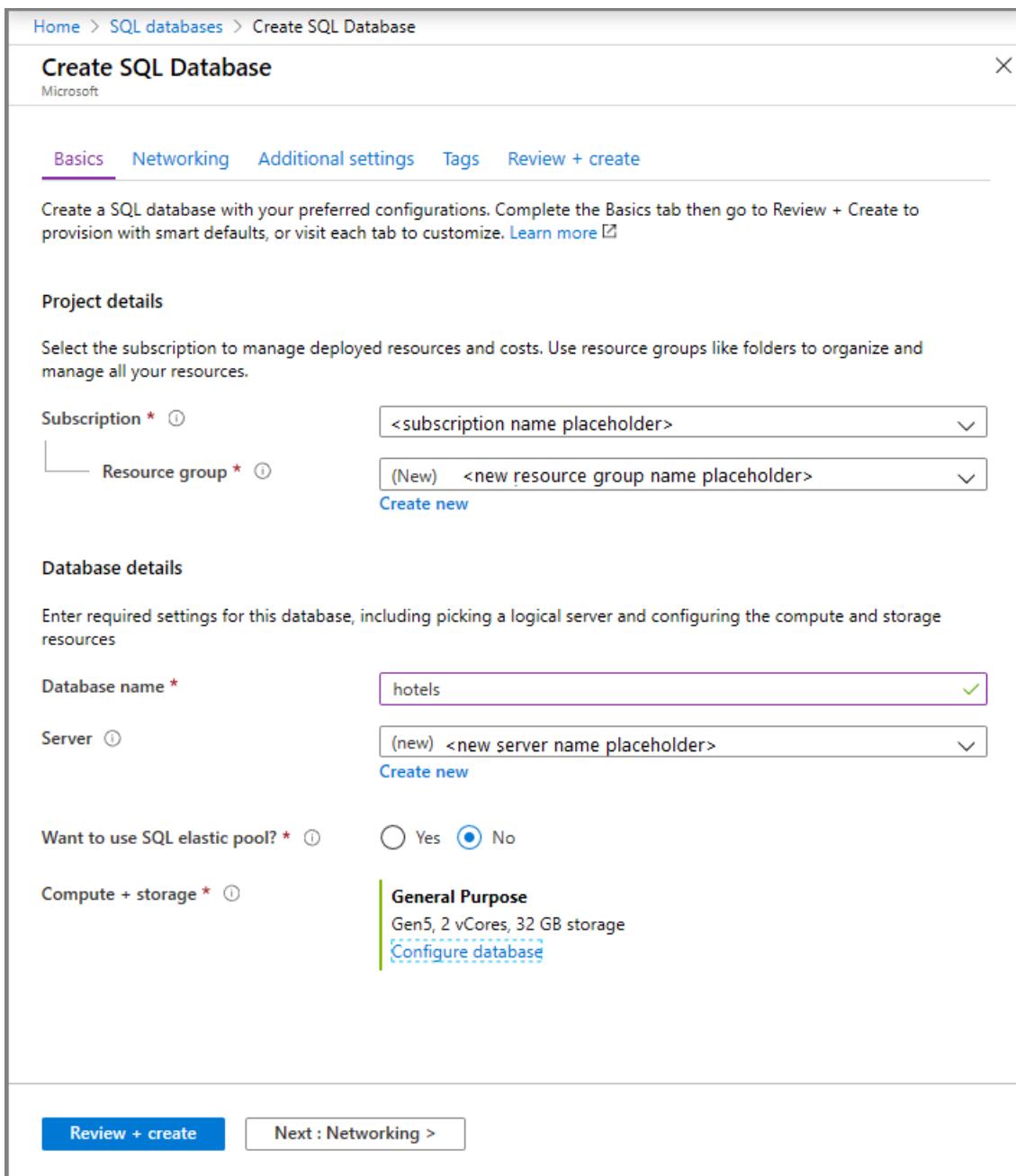
Database name * [▼](#) [Configure database](#)

Server [?](#) [▼](#) [Create new](#)

Want to use SQL elastic pool? * [?](#) Yes No

Compute + storage * [?](#) [General Purpose](#)
Gen5, 2 vCores, 32 GB storage
[Configure database](#)

[Review + create](#) [Next : Networking >](#)



3. Click **Review + create** to deploy the new server and database. Wait for the server and database to deploy.

4. On the navigation pane, click **Query editor (preview)** and enter the user name and password of server admin.

If access is denied, copy the client IP address from the error message, and then click the **Set server firewall** link to add a rule that allows access from your client computer, using your client IP for the range. It can take several minutes for the rule to take effect.

5. In Query editor, click **Open query** and navigate to the location of *hotels.sql* file on your local computer.

6. Select the file and click **Open**. The script should look similar to the following screenshot:

The screenshot shows a SQL query editor interface. At the top, there are tabs for 'Query 1' and 'Query 2', with 'Query 2' being the active tab. Below the tabs are buttons for 'Run', 'Cancel query', 'Save query', 'Export data as json', and more. The main area contains the following SQL code:

```

1 ALTER DATABASE CURRENT
2 SET CHANGE_TRACKING = ON
3 (CHANGE_RETENTION = 2 DAYS, AUTO_CLEANUP = ON)
4
5 CREATE TABLE Hotels (
6     [HotelId] nvarchar(450) NOT NULL PRIMARY KEY,
7     [BaseRate] float NULL,
8     [Category] nvarchar(max) NULL,
9     [Description] nvarchar(max) NULL,
10    [Description_fr] nvarchar(max) NULL,
11    [HotelName] nvarchar(max) NULL,
12    [Tags] nvarchar(max) NULL,
13    [IsDeleted] bit NOT NULL,
14    [LastRenovationDate] DateTime NULL,

```

Below the code, there are two buttons: 'Results' and 'Messages'. The 'Messages' button is highlighted with a blue border. In the bottom right corner of the editor, the message 'Query succeeded: Affected rows: 0Affected rows: 3' is displayed.

7. Click **Run** to execute the query. In the Results pane, you should see a query succeeded message, for 3 rows.
8. To return a rowset from this table, you can execute the following query as a verification step:

```
SELECT * FROM Hotels
```

9. Copy the ADO.NET connection string for the database. Under **Settings > Connection Strings**, copy the ADO.NET connection string, similar to the example below.

```
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;
```

You will need this connection string in the next exercise, setting up your environment.

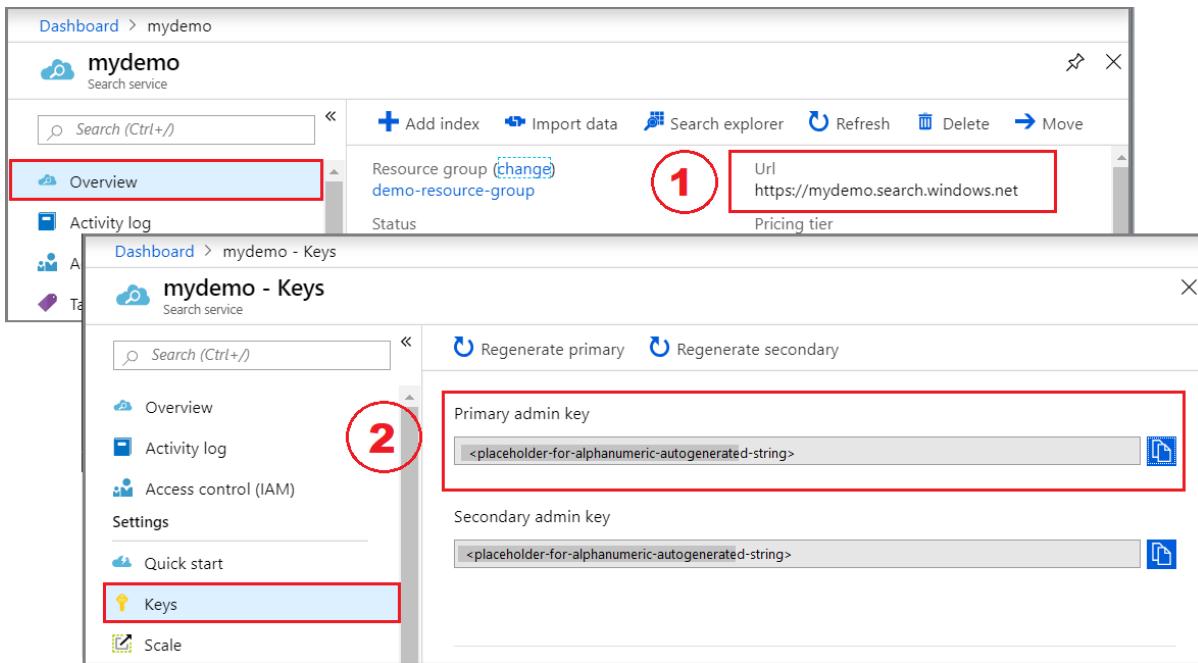
Azure Cognitive Search

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 `searchServiceName`, if the full URL is "https://my-demo-service.search.windows.net", the service name to provide is "my-demo-service".
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;"
```

```
{
  "SearchServiceName": "<placeholder-Azure-Search-service-name>",
  "SearchServiceAdminApiKey": "<placeholder-admin-key-for-Azure-Search>",
  "AzureSqlConnectionString": "<placeholder-ADO.NET-connection-string>",
}
```

5. In the connection string, make sure the connection string contains 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`.

```
    . . .
[IsSearchable, IsFilterable, IsSortable]
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 a 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 leveraging the built-in [change detection features](#) of Azure SQL. The 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.

```
Console.WriteLine("Creating data source...");

DataSource dataSource = DataSource.AzureSql(
    name: "azure-sql",
    sqlConnectionString: configuration["AzureSQLConnectionString"],
    tableOrViewName: "hotels",
    deletionDetectionPolicy: new SoftDeleteColumnDeletionDetectionPolicy(
        softDeleteColumnName: "IsDeleted",
        softDeleteMarkerValue: "true"));
dataSource.DataChangeDetectionPolicy = new SqlIntegratedChangeTrackingPolicy();

searchService.DataSources.CreateOrUpdateAsync(dataSource).Wait();
```

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.

```

Console.WriteLine("Creating Azure SQL indexer...");
Indexer indexer = new Indexer(
    name: "azure-sql-indexer",
    dataSourceName: dataSource.Name,
    targetIndexName: index.Name,
    schedule: new IndexingSchedule(TimeSpan.FromDays(1)));
// Indexers contain metadata about how much they have already indexed
// If we already ran the sample, the indexer will remember that it already
// indexed the sample data and not run again
// To avoid this, reset the indexer if it exists
exists = await searchService.Indexers.ExistsAsync(indexer.Name);
if (exists)
{
    await searchService.Indexers.ResetAsync(indexer.Name);
}

await searchService.Indexers.CreateOrUpdateAsync(indexer);

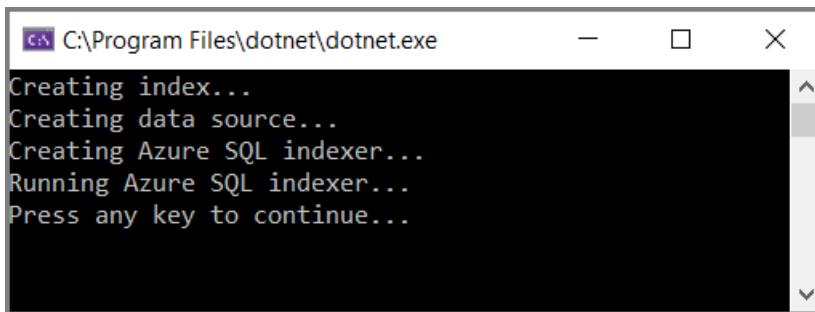
// We created the indexer with a schedule, but we also
// want to run it immediately
Console.WriteLine("Running Azure SQL indexer...");

try
{
    await searchService.Indexers.RunAsync(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 limited to the service name in this tutorial. If you entered the full URL, 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 access.
- Resource limits. Recall that the Free tier has limits of 3 indexes, indexers, and data sources. A service at the maximum limit cannot 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. Select the hotels index. On the hotels page, **Search explorer** is the first tab.
3. Click **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

10/4/2020 • 9 minutes to read • [Edit Online](#)

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 desktop 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.



Storage account - blob, file, table,
queue
[Quickstarts + tutorials](#)

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.

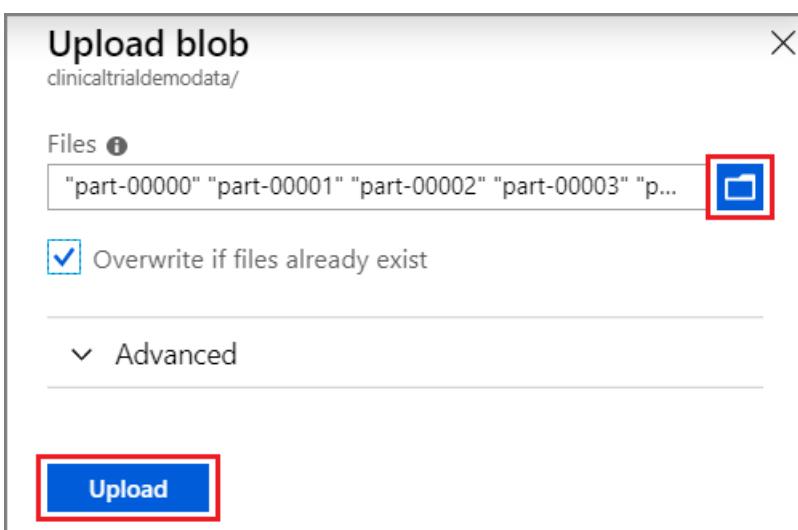
6. Click **Blobs** service.

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.

The screenshot shows the Azure Cognitive Search service 'mydemo'. In the main pane, the 'Overview' link is highlighted with a red box. The 'Url' field, which contains the endpoint 'https://mydemo.search.windows.net', is also highlighted with a red box and has a red circle with the number '1' above it. In the sub-pane, the 'Keys' link is highlighted with a red box. The 'Primary admin key' input field, which contains the placeholder text '<placeholder-for-alphanumeric-autogenerated-string>', is highlighted with a red box and has a red circle with the number '2' above it.

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 [Explore Azure Cognitive Search REST APIs using Postman](#).

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.

The screenshot shows a Postman request configuration. The method is set to 'GET' and the URL is 'https://mydemo.search.windows.net/indexes?api-version=2020-06-30'. The 'Headers' tab is selected, showing two entries: 'Content-Type: application/json' and 'api-key: <placeholder-api-key-for-your-service>'. The response status is shown as 'Status: 200 OK' and 'Time: 540 ms'.

URLs 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.

```
{  
    "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:

```
{  
    "@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": "[mycontainernamewhere]",  
        "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.

```
{
  "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, "facetable": false, "filterable": true, "sortable": false},
    {"name": "metadata_storage_size", "type": "Edm.String", "searchable": false, "retrievable": true, "facetable": true, "filterable": true, "sortable": false},
    {"name": "metadata_content_type", "type": "Edm.String", "searchable": true, "retrievable": true, "facetable": false, "filterable": true, "sortable": false}
  ]
}
```

- Send the request. The response should look like:

```
{
    "@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](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.

```
{
    "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:

```
{  
    "@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:

```
{
    "@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.

The screenshot shows the Azure Cognitive Search UI with a GET request to the specified URL. The response body displays two documents, each with line numbers 3 and 4. Line 3 contains the '@odata.count' field with the value '100'. Line 4 contains the 'value' field, which is an array of two objects. Each object has an '@search.score' field (1.0) and a 'metadata_storage_size' field with values '33060' and '34219' respectively. The 'Gender' field is also present in both objects with the value 'Both'.

```

3   "@odata.count": 100,
4   "value": [
5       {
6           "@search.score": 1.0,
7           "Gender": "Both",
8           "metadata_storage_size": "33060"
9       },
10      {
11          "@search.score": 1.0,
12          "Gender": "Both",
13          "metadata_storage_size": "34219"
14      },
15  ]

```

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.

```
3 "@odata.count": 10,
4 "value": [
5   {
6     "@search.score": 1.0,
7     "FileName": "NCT00000161.txt",
8     "MinimumAge": 45,
9     "Title": "Randomized Trials of Vitamin Supplements and Eye Disease",
10    "MyURL": "https://azure.storagedemos.com/clinical-trials/NCT00000161.txt",
11    "Gender": "Female",
12    "MaximumAge": 0,
13    "Summary": "To determine whether vitamin E supplementation reduces the risk of cataract and age-related macular degeneration in women. To determine whether vitamin C supplementation reduces the risk of cataract and AMD in women. To determine whether beta-carotene supplementation reduces the risk of cataract and AMD in women. To determine whether alternative treatments reduce the risk of cataract and AMD in women. To identify potential risk factors for cataract and AMD including alcohol intake, blood pressure, blood cholesterol, cardiovascular disease, height, body mass index, and dietary factors.", "NCTID": "NCT00000161",
14    "Phase": "Phase 3",
15    "Date": "ClinicalTrials.gov processed this data on October 25, 2016",
16    "OverallStatus": "Active, not recruiting",
17    "OrgStudyId": "NEI-63",
18    "HealthyVolunteers": "Accepts Healthy Volunteers",
19    "Keywords": [
20      "Age-Related Macular Degeneration"
21    ],
22  }
]
```

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.

```
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

10/4/2020 • 13 minutes to read • [Edit Online](#)

Azure Cognitive Search can import, analyze, and index data from multiple data sources into a single consolidate search index. This supports situations where structured data is aggregated with less-structured or even plain text data from other sources, like text, HTML, or JSON documents.

This tutorial describes how to index hotel data from an Azure Cosmos DB data source and merge that with hotel room details drawn from Azure Blob Storage documents. The result will be a combined hotel search index containing complex data types.

This tutorial uses C# and the [.NET SDK](#). 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.

Prerequisites

- [Azure Cosmos DB](#)
- [Azure Storage](#)
- [Visual Studio 2019](#)
- [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 [azure-search-dotnet-samples](#) GitHub repository, in the [multiple-data-sources](#) folder.

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.

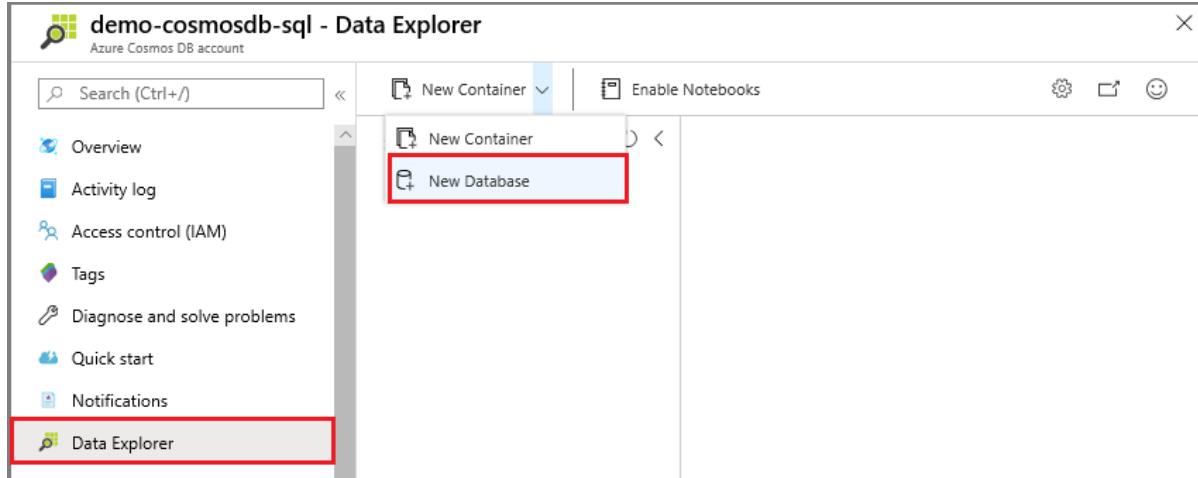
If possible, create all services in the same region and resource group for proximity and manageability. In practice, your services can be in any region.

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 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.

The screenshot shows the 'New Database' configuration dialog. It has fields for 'Database id' (containing 'hotel-rooms-db'), 'Provision throughput' (checkbox checked), 'Throughput (400 - 100,000 RU/s)' (400 is entered), and 'Autopilot (preview)' or 'Manual' (Manual is selected). At the bottom is an 'OK' button.

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.

Add Container

* Database id ⓘ
 Create new Use existing
 hotel-rooms-db

* Container id ⓘ

* Partition key ⓘ

My partition key is larger than 100 bytes

Provision dedicated throughput for this container ⓘ

Unique keys ⓘ
 + Add unique key

OK

5. Select **Items** under **hotels**, and then click **Upload Item** on the command bar. Navigate to and then select the file **cosmosdb/HotelsDataSubset_CosmosDb.json** in the project folder.

SQL API

hotel-rooms-db

hotels

Items

Upload Item

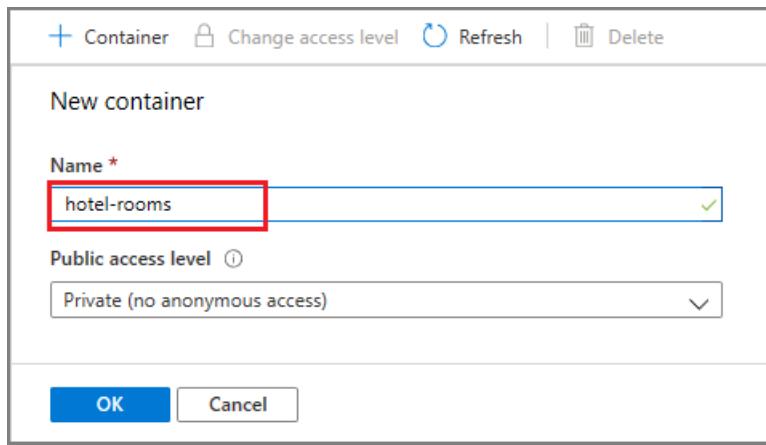
Select JSON Files ⓘ
 "HotelsDataSubset_CosmosDb.json"

Upload

6. Use the Refresh button to refresh your view of the items in the hotels collection. You should see seven new database documents listed.

Azure Blob storage

- Sign in to the [Azure portal](#), navigate to your Azure storage account, click **Blobs**, and then click **+ Container**.
- 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 click **Upload**.

The screenshot displays the Azure Storage Explorer interface. On the left, the 'hotel-rooms' container is shown with an 'Upload' button highlighted by a red box. The main pane shows a table with columns: Name, Modified, Access tier, and Blob type. A message at the bottom states 'No blobs found.' On the right, a modal window titled 'Upload blob' is open, showing a list of files: 'Rooms1.json', 'Rooms10.json', 'Rooms11.json', and 'Roo...'. This list is also highlighted with a red box. Below the file list are checkboxes for 'Overwrite if files already exist' and 'Advanced' settings, followed by a large blue 'Upload' button.

After the upload completes, the files should appear in the list for the data container.

Azure Cognitive Search

The third 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

To interact with your Azure Cognitive Search service you will need 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.

Get the query key as well. It's a best practice to issue query requests with read-only access.

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 2019 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 **Microsoft.Azure.Search** (version 9.0.1, or later). You will have to click through additional dialogs to complete the installation.

3. Search for the **Microsoft.Extensions.Configuration.Json** NuGet package and install it as well.

4. Open the solution file `AzureSearchMultipleDataSources.sln`.
5. In Solution Explorer, edit the `appsettings.json` file to add connection information.

```
{
  "SearchServiceName": "Put your search service name here",
  "SearchServiceAdminApiKey": "Put your primary or secondary API key here",
  "BlobStorageAccountName": "Put your Azure Storage account name here",
  "BlobStorageConnectionString": "Put your Azure Blob Storage connection string here",
  "CosmosDBConnectionString": "Put your Cosmos DB connection string here",
  "CosmosDBDatabaseName": "hotel-rooms-db"
}
```

The first two entries use the URL and admin keys for your Azure Cognitive Search service. Given an endpoint of <https://mydemo.search.windows.net>, for example, the service name to provide is `mydemo`.

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 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 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 by mapping the `Id` field from the blobs to the `HotelId` key field in the index.

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 `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 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 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 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 is 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.

```
...
[IsSearchable, IsFilterable, IsSortable]
public string HotelName { get; set; }
...
public Room[] Rooms { get; set; }
...
```

In the **Program.cs** file, the index is defined with a name and a field collection generated by the `FieldBuilder.BuildForType<Hotel>()` method, and then created as follows:

```
private static async Task CreateIndex(string indexName, SearchServiceClient searchService)
{
    // 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.
    var definition = new Index()
    {
        Name = indexName,
        Fields = FieldBuilder.BuildForType<Hotel>()
    };
    await searchService.Indexes.CreateAsync(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 the data source object, including settings specific to Azure Cosmos DB sources, such as the `[useChangeDetection]` property.

```

private static async Task CreateAndRunCosmosDbIndexer(string indexName, SearchServiceClient searchService)
{
    // Append the database name to the connection string
    string cosmosConnectionString =
        configuration["CosmosDBConnectionString"]
        + ";Database="
        + configuration["CosmosDBDatabaseName"];

    DataSource cosmosDbDataSource = DataSource.CosmosDb(
        name: configuration["CosmosDBDatabaseName"],
        cosmosDbConnectionString: cosmosConnectionString,
        collectionName: "hotels",
        useChangeDetection: true);

    // 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 searchService.DataSources.CreateOrUpdateAsync(cosmosDbDataSource);
}

```

After the data source is created, the program sets up an Azure Cosmos DB indexer named **hotel-rooms-cosmos-indexer**.

```

Indexer cosmosDbIndexer = new Indexer(
    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 this sample, the indexer will remember that it already
// indexed the sample data and not run again.
// To avoid this, reset the indexer if it exists.
bool exists = await searchService.Indexers.ExistsAsync(cosmosDbIndexer.Name);
if (exists)
{
    await searchService.Indexers.ResetAsync(cosmosDbIndexer.Name);
}
await searchService.Indexers.CreateOrUpdateAsync(cosmosDbIndexer);

```

The program will delete any existing indexers with the same name before creating the new one, in case you want to run this example more than once.

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.

Index Azure Cosmos DB data

Once the data source and the indexer have been created, the code that runs the indexer is brief:

```

try
{
    await searchService.Indexers.RunAsync(cosmosDbIndexer.Name);
}
catch (CloudException e) when (e.Response.StatusCode == (HttpStatusCode)429)
{
    Console.WriteLine("Failed to run indexer: {0}", e.Response.Content);
}

```

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 contained no 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.

```
private static async Task CreateAndRunBlobIndexer(string indexName, SearchServiceClient searchService)
{
    DataSource blobDataSource = DataSource.AzureBlobStorage(
        name: configuration["BlobStorageAccountName"],
        storageConnectionString: configuration["BlobStorageConnectionString"],
        containerName: "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 searchService.DataSources.CreateOrUpdateAsync(blobDataSource);
```

After the data source is created, the program sets up a blob indexer named **hotel-rooms-blob-indexer**.

```
// Add a field mapping to match the Id field in the documents to
// the HotelId key field in the index
List<FieldMapping> map = new List<FieldMapping> {
    new FieldMapping("Id", "HotelId")
};

Indexer blobIndexer = new Indexer(
    name: "hotel-rooms-blob-indexer",
    dataSourceName: blobDataSource.Name,
    targetIndexName: indexName,
    fieldMappings: map,
    parameters: new IndexingParameters().ParseJson(),
    schedule: new IndexingSchedule(TimeSpan.FromDays(1)));

// Reset the indexer if it already exists
bool exists = await searchService.Indexers.ExistsAsync(blobIndexer.Name);
if (exists)
{
    await searchService.Indexers.ResetAsync(blobIndexer.Name);
}
await searchService.Indexers.CreateOrUpdateAsync(blobIndexer);
```

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 parameters that identify the parsing mode to be used. The parsing mode differs for blobs that represent a single document, or multiple documents within the same blob. In this example, each blob represents a single index document, so the code uses the `IndexingParameters.ParseJson()` parameter.

For more information about indexer parsing parameters for JSON blobs, see [Index JSON blobs](#). For more information about specifying these parameters using the .NET SDK, see the [IndexerParametersExtension](#) class.

The program will delete any existing indexers with the same name before creating the new one, in case you want to run this example more than once.

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.

Index blob data

Once the Blob storage data source and indexer have been created, the code that runs the indexer is simple:

```

try
{
    await searchService.Indexers.RunAsync(cosmosDbIndexer.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.

Usage	Monitoring	Indexes	Indexers	Data sources	Skillsets
NAME	DOCUMENT COUNT			STORAGE SIZE	
hotel-rooms-sample	7			114.83 kB	

Click on the **hotel-rooms-sample** index in the list. You will 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 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 concept of ingesting data from multiple sources, let's take a closer look at indexer configuration, starting with Cosmos DB.

[Configure an Azure Cosmos DB indexer](#)

Tutorial: Optimize indexing with the push API

10/4/2020 • 12 minutes to read • [Edit Online](#)

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 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 data asynchronously
- Use multiple threads to increase indexing speeds
- Use an exponential backoff retry strategy to retry failed items

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 [optimize-data-indexing](#) 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.

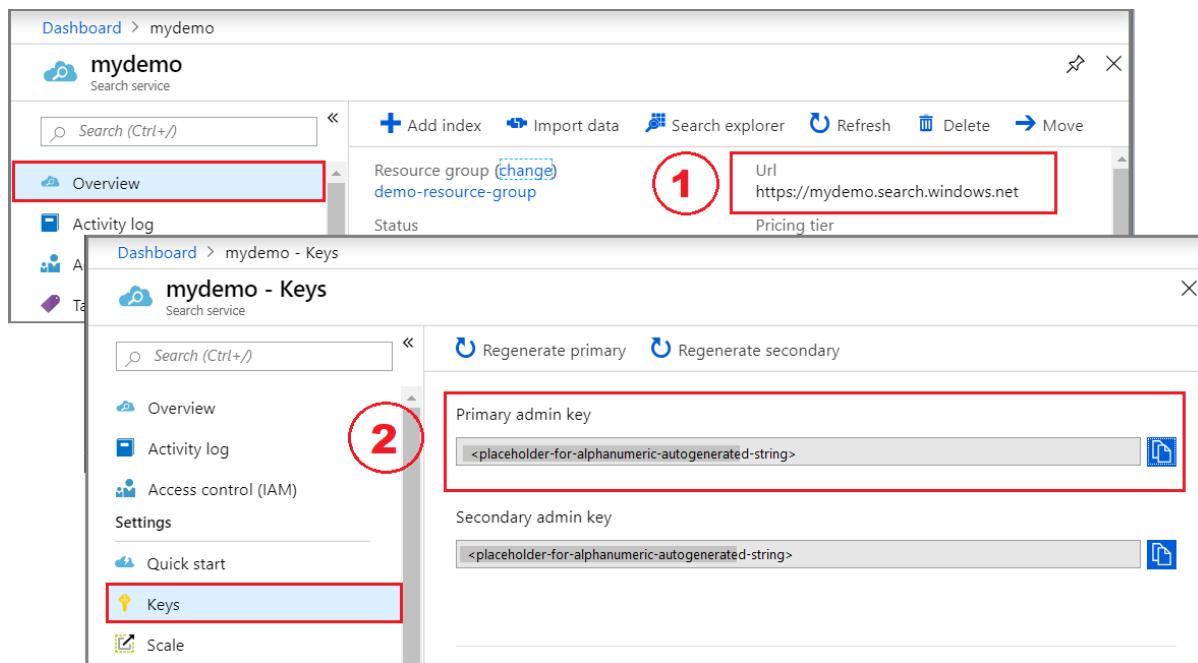
1 - Create Azure Cognitive Search service

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.
3. For `searchServiceName`, if the full URL is "https://my-demo-service.search.windows.net", the service name to provide is "my-demo-service".

```
{  
  "SearchServiceName": "<YOUR-SEARCH-SERVICE-NAME>",  
  "SearchServiceAdminApiKey": "<YOUR-ADMIN-API-KEY>",  
  "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.

```
    ...
    [IsSearchable, IsSortable]
    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.BuildForType<Hotel>\(\)](#) method, and then created as follows:

```
private static async Task CreateIndex(string indexName, SearchServiceClient searchService)
{
    // 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.
    var definition = new Index()
    {
        Name = indexName,
        Fields = FieldBuilder.BuildForType<Hotel>()
    };
    await searchService.Indexes.CreateAsync(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 two lines of code:

```
DataGenerator dg = new DataGenerator();
List<Hotel> hotels = dg.GetHotels(100000, "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 \(REST API\)](#)
- [indexAction class](#) or [indexBatch 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.

```

public static async Task TestBatchSizes(ISearchIndexClient indexClient, 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 UploadDocuments(indexClient, hotels);
            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 {1} \t {2} \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);
    }
}

```

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.

```

public static double EstimateObjectSize(object data)
{
    // converting data to json for more accurate sizing
    var json = JsonConvert.SerializeObject(data);

    // converting object to byte[] to determine the size of the data
    BinaryFormatter bf = new BinaryFormatter();
    MemoryStream ms = new MemoryStream();
    byte[] Array;

    bf.Serialize(ms, json);
    Array = ms.ToArray();

    // converting from bytes to megabytes
    double sizeInMb = (double)Array.Length / 1000000;

    return sizeInMb;
}

```

The function requires an `ISearchIndexClient` 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.

```
await TestBatchSizes(indexClient, 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:

```
// Create batch of documents for indexing
IndexBatch<Hotel> batch = IndexBatch.Upload(hotels);

// Define parameters for exponential backoff
int attempts = 0;
TimeSpan delay = delay = TimeSpan.FromSeconds(2);
int maxRetryAttempts = 5;
```

It's important to catch `IndexBatchException` as these exceptions indicates that the indexing operation only partially succeeded (207s). Failed items should be retried using the `FindFailedActionsToRetry` method that makes it easy to create a new batch containing only the failed items.

Exceptions other than `IndexBatchException` should also be caught and indicate the request failed completely. These exceptions are less common, particularly with the .NET SDK as it retries 503s automatically.

```
// Implement exponential backoff
do
{
    try
    {
        attempts++;
        var response = await indexClient.Documents.IndexAsync(batch);
        break;
    }
    catch (IndexBatchException ex)
    {
        Console.WriteLine("[Attempt: {0} of {1} Failed] - Error: {2}", attempts, maxRetryAttempts,
ex.Message);

        if (attempts == maxRetryAttempts)
            break;

        // Find the failed items and create a new batch to retry
        batch = ex.FindFailedActionsToRetry(batch, x => x.HotelId);
        Console.WriteLine("Retrying failed documents using exponential backoff...\n");

        Task.Delay(delay).Wait();
        delay = delay * 2;
    }
    catch (Exception ex)
    {
        Console.WriteLine("[Attempt: {0} of {1} Failed] - Error: {2} \n", attempts, maxRetryAttempts,
ex.Message);

        if (attempts == maxRetryAttempts)
            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:

```
ExponentialBackoff.IndexData(indexClient, hotels, 1000, 8).Wait();
```

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 37000:
[Attempt: 1 of 5 Failed] - Error: 71 of 1000 indexing actions in the batch failed. The remaining actions succeeded and modified the index. Check the IndexResponse property for the status of each index action.
[Status Code 503] - 71 documents
[Status Code 201] - 929 documents
Retrying failed documents using exponential backoff...
```

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 programmatically or using the [Search explorer](#) in the portal.

Programmatically

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:

```
long indexDocCount = indexClient.Documents.Count();
```

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.

```
IndexGetStatisticsResult indexStats = serviceClient.Indexes.GetStatistics(configuration["SearchIndexName"]);
```

Azure portal

In Azure portal, open the search service **Overview** page, and find the **optimize-indexing** index in the **Indexes** list.

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](#)

Tutorial: AI-generated searchable content from Azure blobs using the .NET SDK

10/4/2020 • 23 minutes to read • [Edit Online](#)

If you have unstructured text or images in Azure Blob storage, an [AI enrichment pipeline](#) can extract information and create new content that is useful for full-text search or knowledge mining scenarios. In this C# tutorial, apply Optical Character Recognition (OCR) on images and perform natural language processing to create new fields that you can leverage in queries, facets, and filters.

This tutorial uses C# and the [.NET SDK](#) to perform the following tasks:

- Start with application files and images in Azure Blob storage.
- Define a pipeline to add OCR, text extraction, language detection, entity and key phrase recognition.
- Define an index to store the output (raw content, plus pipeline-generated name-value pairs).
- Execute the pipeline to start transformations and analysis, and 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 Storage](#)
- [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

1. Open this [OneDrive folder](#) and on the top-left corner, click **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. You'll use 7 for this exercise.

You can also download the source code for this tutorial. Source code is in the `tutorial-ai-enrichment` 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. 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. Click + Container to create a container and name it *cog-search-demo*.
8. Select *cog-search-demo* and then click **Upload** to open the folder where you saved the download files. Select all fourteen files and click **OK** to upload.

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

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 *blobstoragewestus* 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:

```
DefaultEndpointsProtocol=https;AccountName=blobstoragewestus;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 Text Analytics 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

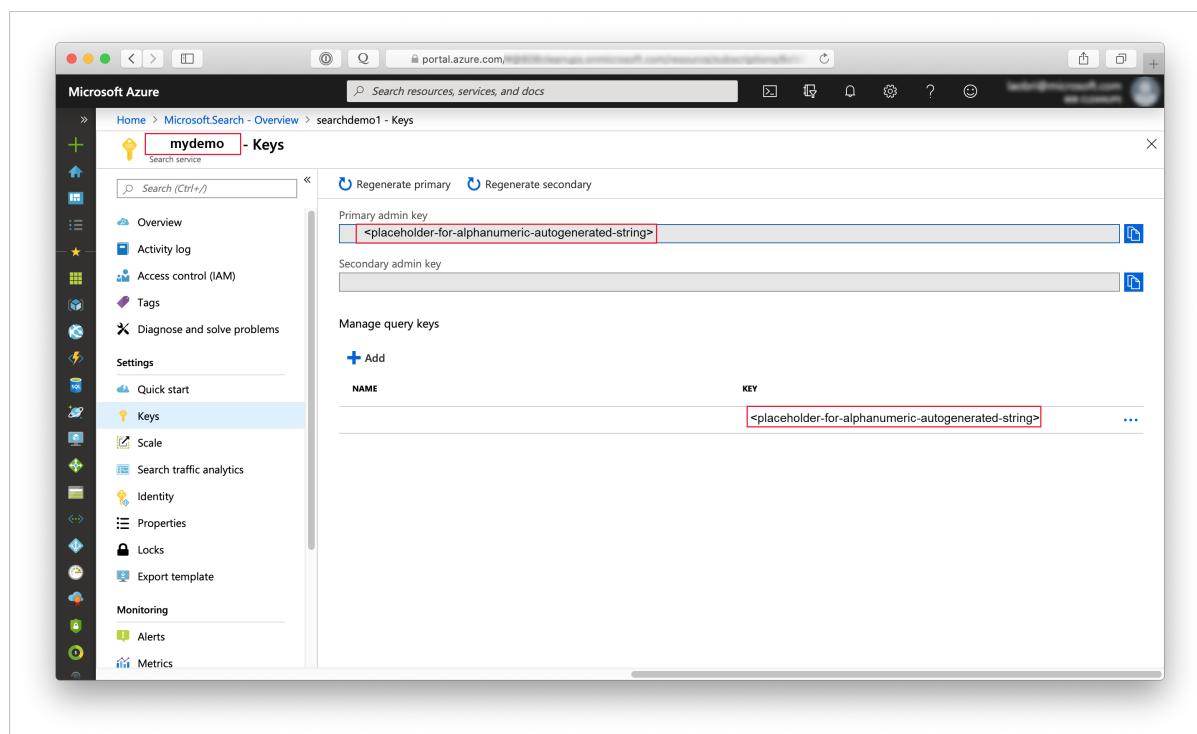
The third component is Azure Cognitive Search, which you can [create in the portal](#). You can use the Free tier to complete this walkthrough.

Get 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. 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.

Get the query key as well. It's a best practice to issue query requests with read-only access.



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 NuGet packages

The [Azure Cognitive Search .NET SDK](#) consists of a few client libraries that enable 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. These client libraries are all distributed as NuGet packages.

For this project, install version 9 or later of the `Microsoft.Azure.Search` NuGet package.

1. In a browser, go to [Microsoft.Azure.Search NuGet package page](#).
2. Select the latest version (9 or later).
3. Copy the Package Manager command.
4. Open the Package Manager Console. Select **Tools > NuGet Package Manager > Package Manager Console**.
5. Paste and run the command that you copied in the previous step.

Next, install the latest `Microsoft.Extensions.Configuration.Json` NuGet package.

1. Select **Tools > NuGet Package Manager > Manage NuGet Packages for Solution....**
2. Click **Browse** and search for the `Microsoft.Extensions.Configuration.Json` NuGet package.
3. Select the package, select your project, confirm the version is the latest stable version, then click **Install**.

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.

```
{  
  "SearchServiceName": "Put your search service name 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 **SearchServiceName**, enter the short service name and not the full URL.

Add namespaces

In `Program.cs`, add the following namespaces.

```
using System;
using System.Collections.Generic;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Microsoft.Extensions.Configuration;

namespace EnrichwithAI
```

Create a client

Create an instance of the `SearchServiceClient` class under `Main`.

```
public static void Main(string[] args)
{
    // Create service client
    IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.json");
    IConfigurationRoot configuration = builder.Build();
    SearchServiceClient serviceClient = CreateSearchServiceClient(configuration);
```

`CreateSearchServiceClient` creates a new `SearchServiceClient` using values that are stored in the application's config file (`appsettings.json`).

```
private static SearchServiceClient CreateSearchServiceClient(IConfigurationRoot configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string adminApiKey = configuration["SearchServiceAdminApiKey"];

    SearchServiceClient serviceClient = new SearchServiceClient(searchServiceName, new
    SearchCredentials(adminApiKey));
    return serviceClient;
}
```

NOTE

The `SearchServiceClient` class manages connections to your search service. In order to avoid opening too many connections, you should try to share a single instance of `SearchServiceClient` in your application if possible. Its 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.

```
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

`SearchServiceClient` has a `DataSources` property. This property provides all the methods you need to create, list, update, or delete Azure Cognitive Search data sources.

Create a new `DataSource` instance by calling `serviceClient.DataSources.CreateOrUpdate(dataSource)`.

`DataSource.AzureBlobStorage` requires that you specify the data source name, connection string, and blob container name.

```
private static DataSource CreateOrUpdateDataSource(SearchServiceClient serviceClient, IConfigurationRoot configuration)
{
    DataSource dataSource = DataSource.AzureBlobStorage(
        name: "demodata",
        storageConnectionString: configuration["AzureBlobConnectionString"],
        containerName: "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
    {
        serviceClient.DataSources.CreateOrUpdate(dataSource);
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to create or update the data source\n Exception message: {0}\n", e.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.

```
public static void Main(string[] args)
{
    // Create service client
    IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.json");
    IConfigurationRoot configuration = builder.Build();
    SearchServiceClient serviceClient = CreateSearchServiceClient(configuration);

    // Create or Update the data source
    Console.WriteLine("Creating or updating the data source...");
    DataSource dataSource = CreateOrUpdateDataSource(serviceClient, 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 dashboard page, verify the Data Sources tile has a new item. You might need to wait a few minutes for the portal page to refresh.

The screenshot shows the 'Data sources' section of the Azure Cognitive Search portal. It lists three data sources: 'demodata' (selected), 'hotels-sample', and 'realestate-us-sample'. Each entry includes a small icon representing the source type (CSV, JSON, XML, etc.) and its name.

Type ↑↓	Name ↑↓	Table/Collection ↑↓
CSV	demodata	cog-search-demo
JSON	hotels-sample	hotels
SQL	realestate-us-sample	Listings_5K_KingCounty_WA

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 field.
- [Language Detection](#) to identify the content's language.
- [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.
- [Entity Recognition](#) for extracting the names of organizations from content in the blob container.
- [Key Phrase Extraction](#) to pull out the top key phrases.

During initial 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, set the input as `"/document/content"` to use this text.

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 OCR skill 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"`.

```
private static OcrSkill CreateOcrSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>
    {
        new InputFieldMappingEntry(
            name: "image",
            source: "/document/normalized_images/*")
    };

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>
    {
        new OutputFieldMappingEntry(
            name: "text",
            targetName: "text")
    };

    OcrSkill ocrSkill = new OcrSkill(
        description: "Extract text (plain and structured) from image",
        context: "/document/normalized_images/*",
        inputs: inputMappings,
        outputs: outputMappings,
        defaultLanguageCode: OcrSkillLanguage.En,
        shouldDetectOrientation: true);

    return ocrSkill;
}
```

Merge skill

In this section you'll create a **Merge** skill that merges the document content field with the text that was produced by the OCR skill.

```

private static MergeSkill CreateMergeSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>
    {
        new InputFieldMappingEntry(
            name: "text",
            source: "/document/content"),
        new InputFieldMappingEntry(
            name: "itemsToInsert",
            source: "/document/normalized_images/*/text"),
        new InputFieldMappingEntry(
            name: "offsets",
            source: "/document/normalized_images/*/contentOffset")
    };

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>
    {
        new OutputFieldMappingEntry(
            name: "mergedText",
            targetName: "merged_text")
    };

    MergeSkill mergeSkill = new 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",
        inputs: inputMappings,
        outputs: outputMappings,
        insertPreTag: " ",
        insertPostTag: " ");
}

return mergeSkill;
}

```

Language detection skill

The **Language Detection** skill 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.

```

private static LanguageDetectionSkill CreateLanguageDetectionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>
    {
        new InputFieldMappingEntry(
            name: "text",
            source: "/document/merged_text")
    };

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>
    {
        new OutputFieldMappingEntry(
            name: "languageCode",
            targetName: "languageCode")
    };

    LanguageDetectionSkill languageDetectionSkill = new LanguageDetectionSkill(
        description: "Detect the language used in the document",
        context: "/document",
        inputs: inputMappings,
        outputs: outputMappings);

    return languageDetectionSkill;
}

```

Text split skill

The below `Split` skill 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`.

```
private static SplitSkill CreateSplitSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>
    {
        new InputFieldMappingEntry(
            name: "text",
            source: "/document/merged_text"),
        new InputFieldMappingEntry(
            name: "languageCode",
            source: "/document/languageCode")
    };

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>
    {
        new OutputFieldMappingEntry(
            name: "textItems",
            targetName: "pages")
    };

    SplitSkill splitSkill = new SplitSkill(
        description: "Split content into pages",
        context: "/document",
        inputs: inputMappings,
        outputs: outputMappings,
        textSplitMode: TextSplitMode.Pages,
        maximumPageLength: 4000);

    return splitSkill;
}
```

Entity recognition skill

This `EntityRecognitionSkill` instance is set to recognize category type `organization`. The **Entity Recognition** skill 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"`.

```

private static EntityRecognitionSkill CreateEntityRecognitionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>
    {
        new InputFieldMappingEntry(
            name: "text",
            source: "/document/pages/*")
    };

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>
    {
        new OutputFieldMappingEntry(
            name: "organizations",
            targetName: "organizations")
    };

    List<EntityCategory> entityCategory = new List<EntityCategory>
    {
        EntityCategory.Organization
    };

    EntityRecognitionSkill entityRecognitionSkill = new EntityRecognitionSkill(
        description: "Recognize organizations",
        context: "/document/pages/*",
        inputs: inputMappings,
        outputs: outputMappings,
        categories: entityCategory,
        defaultLanguageCode: EntityRecognitionSkillLanguage.En);

    return entityRecognitionSkill;
}

```

Key phrase extraction skill

Like the `EntityRecognitionSkill` instance that was just created, the **Key Phrase Extraction** skill is called for each page of the document.

```

private static KeyPhraseExtractionSkill CreateKeyPhraseExtractionSkill()
{
    List<InputFieldMappingEntry> inputMappings = new List<InputFieldMappingEntry>();
    inputMappings.Add(new InputFieldMappingEntry(
        name: "text",
        source: "/document/pages/*"));
    inputMappings.Add(new InputFieldMappingEntry(
        name: "languageCode",
        source: "/document/languageCode"));

    List<OutputFieldMappingEntry> outputMappings = new List<OutputFieldMappingEntry>();
    outputMappings.Add(new OutputFieldMappingEntry(
        name: "keyPhrases",
        targetName: "keyPhrases"));

    KeyPhraseExtractionSkill keyPhraseExtractionSkill = new KeyPhraseExtractionSkill(
        description: "Extract the key phrases",
        context: "/document/pages/*",
        inputs: inputMappings,
        outputs: outputMappings);

    return keyPhraseExtractionSkill;
}

```

Build and create the skillset

Build the `Skillset` using the skills you created.

```

private static Skillset CreateOrUpdateDemoSkillSet(SearchServiceClient serviceClient, IList<Skill> skills)
{
    Skillset skillset = new Skillset(
        name: "demoskillset",
        description: "Demo skillset",
        skills: skills);

    // 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
    {
        serviceClient.Skillsets.CreateOrUpdate(skillset);
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to create the skillset\n Exception message: {0}\n", e.Message);
        ExitProgram("Cannot continue without a skillset");
    }

    return skillset;
}

```

Add the following lines to `Main`.

```

// 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<Skill> skills = new List<Skill>
{
    ocrSkill,
    mergeSkill,
    languageDetectionSkill,
    splitSkill,
    entityRecognitionSkill,
    keyPhraseExtractionSkill
};

Skillset skillset = CreateOrUpdateDemoSkillSet(serviceClient, skills);

```

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
<code>id</code>	Edm.String

FIELD NAMES	FIELD TYPES
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 `Microsoft.Azure.Search` and `Microsoft.Azure.Search.Models` namespaces.

Add the below model class definition to `DemoIndex.cs` and include it in the same namespace where you'll create the index.

```
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;

namespace EnrichwithAI
{
    // The SerializePropertyNamesAsCamelCase attribute is defined in the Azure Search .NET SDK.
    // It ensures that Pascal-case property names in the model class are mapped to camel-case
    // field names in the index.
    [SerializePropertyNamesAsCamelCase]
    public class DemoIndex
    {
        [System.ComponentModel.DataAnnotations.Key]
        [IsSearchable, IsSortable]
        public string Id { get; set; }

        [IsSearchable]
        public string Content { get; set; }

        [IsSearchable]
        public string LanguageCode { get; set; }

        [IsSearchable]
        public string[] KeyPhrases { get; set; }

        [IsSearchable]
        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.

```

private static Index CreateDemoIndex(SearchServiceClient serviceClient)
{
    var index = new Index()
    {
        Name = "demoindex",
        Fields = FieldBuilder.BuildForType<DemoIndex>()
    };

    try
    {
        bool exists = serviceClient.Indexes.Exists(index.Name);

        if (exists)
        {
            serviceClient.Indexes.Delete(index.Name);
        }

        serviceClient.Indexes.Create(index);
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to create the index\n Exception message: {0}\n", e.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`.

```

// Create the index
Console.WriteLine("Creating the index...");
Microsoft.Azure.Search.Models.Index demoIndex = CreateDemoIndex(serviceClient);

```

Add the following using statement to resolve the disambiguate reference.

```
using Index = Microsoft.Azure.Search.Models.Index;
```

To learn more about defining an index, see [Create Index \(Azure Cognitive Search REST API\)](#).

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](#).

```

private static Indexer CreateDemoIndexer(SearchServiceClient serviceClient, DataSource dataSource, Skillset
skillSet, Index index)
{

```

```

    IDictionary<string, object> config = new Dictionary<string, object>();
    config.Add(
        key: "dataToExtract",
        value: "contentAndMetadata");
    config.Add(
        key: "imageAction",
        value: "generateNormalizedImages");

    List<FieldMapping> fieldMappings = new List<FieldMapping>
    {
        new FieldMapping(
            sourceFieldName: "metadata_storage_path",
            targetFieldName: "id",
            mappingFunction: new FieldMappingFunction(
                name: "base64Encode")),
        new FieldMapping(
            sourceFieldName: "content",
            targetFieldName: "content")
    };

    List<FieldMapping> outputMappings = new List<FieldMapping>
    {
        new FieldMapping(
            sourceFieldName: "/document/pages/*/organizations/*",
            targetFieldName: "organizations"),
        new FieldMapping(
            sourceFieldName: "/document/pages/*/keyPhrases/*",
            targetFieldName: "keyPhrases"),
        new FieldMapping(
            sourceFieldName: "/document/languageCode",
            targetFieldName: "languageCode")
    };

    Indexer indexer = new Indexer(
        name: "demoindexer",
        dataSourceName: dataSource.Name,
        targetIndexName: index.Name,
        description: "Demo Indexer",
        skillsetName: skillSet.Name,
        parameters: new IndexingParameters(
            maxFailedItems: -1,
            maxFailedItemsPerBatch: -1,
            configuration: config),
        fieldMappings: fieldMappings,
        outputFieldMappings: outputMappings);

    try
    {
        bool exists = serviceClient.Indexers.Exists(indexer.Name);

        if (exists)
        {
            serviceClient.Indexers.Delete(indexer.Name);
        }

        serviceClient.Indexers.Create(indexer);
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to create the indexer\n Exception message: {0}\n", e.Message);
        ExitProgram("Cannot continue without creating an indexer");
    }

    return indexer;
}

```

Add the following lines to `Main`.

```
// Create the indexer, map fields, and execute transformations  
Console.WriteLine("Creating the indexer and executing the pipeline...");  
Indexer demoIndexer = CreateDemoIndexer(serviceClient, dataSource, skillset, demoIndex);
```

Expect that creating the indexer will take a little 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.

```

private static void CheckIndexerOverallStatus(SearchServiceClient serviceClient, Indexer indexer)
{
    try
    {
        IndexerExecutionInfo demoIndexerExecutionInfo = serviceClient.Indexers.GetStatus(indexer.Name);

        switch (demoIndexerExecutionInfo.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 (Exception e)
    {
        Console.WriteLine("Failed to get indexer overall status\n Exception message: {0}\n", e.Message);
    }
}

```

`IndexerExecutionInfo` 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`.

```

// Check indexer overall status
Console.WriteLine("Check the indexer overall status...");
CheckIndexerOverallStatus(serviceClient, demoIndexer);

```

5 - Search

After indexing is finished, you can run queries that return the contents of individual fields. By default, Azure Cognitive Search returns the top 50 results. The sample data is small so the default works fine. However, when working with larger data sets, you might need to include parameters in the query string to return more results. For instructions, see [How to page results in Azure Cognitive Search](#).

As a verification step, query the index for all of the fields.

Add the following lines to `Main`.

```

DocumentSearchResult<DemoIndex> results;

ISearchIndexClient indexClientForQueries = CreateSearchIndexClient(configuration);

SearchParameters parameters =
    new SearchParameters
    {
        Select = new[] { "organizations" }
    };

try
{
    results = indexClientForQueries.Documents.Search<DemoIndex>("*", parameters);
}
catch (Exception e)
{
    // Handle exception
}

```

`CreateSearchIndexClient` creates a new `SearchIndexClient` using values that are stored in the application's config file (`appsettings.json`). Notice that the search service query API key is used and not the admin key.

```

private static SearchIndexClient CreateSearchIndexClient(IConfigurationRoot configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string queryApiKey = configuration["SearchServiceQueryApiKey"];

    SearchIndexClient indexClient = new SearchIndexClient(searchServiceName, "demoindex", new
    SearchCredentials(queryApiKey));
    return indexClient;
}

```

Add the following code to `Main`. The first try-catch returns the index definition, with the name, type, and attributes of each field. The second is a parameterized query, where `Select` specifies which fields to include in the results, for example `organizations`. A search string of `"*"` returns all contents of a single field.

```

//Verify content is returned after indexing is finished
ISearchIndexClient indexClientForQueries = CreateSearchIndexClient(configuration);

try
{
    results = indexClientForQueries.Documents.Search<DemoIndex>("*");
    Console.WriteLine("First query succeeded with a result count of {0}", results.Results.Count);
}
catch (Exception e)
{
    Console.WriteLine("First query failed\n Exception message: {0}\n", e.Message);
}

SearchParameters parameters =
new SearchParameters
{
    Select = new[] { "organizations" }
};

try
{
    results = indexClientForQueries.Documents.Search<DemoIndex>("*", parameters);
    Console.WriteLine("Second query succeeded with a result count of {0}", results.Results.Count);
}
catch (Exception e)
{
    Console.WriteLine("Second query failed\n Exception message: {0}\n", e.Message);
}

```

Repeat for additional fields: content, languageCode, keyPhrases, and organizations in this exercise. You can return multiple fields via the [Select](#) property using a comma-delimited list.

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

10/4/2020 • 16 minutes to read • [Edit Online](#)

If you have unstructured text or images in Azure Blob storage, an [AI enrichment pipeline](#) can extract information and create new content that is useful for full-text search or knowledge mining scenarios. Although a pipeline can process images, this REST tutorial focuses on text, applying language detection and natural language processing to create new fields that you can leverage in queries, facets, and filters.

This tutorial uses Postman and the [Search REST APIs](#) to perform the following tasks:

- Start with whole documents (unstructured text) such as PDF, HTML, DOCX, and PPTX in Azure Blob storage.
- Define a pipeline that extracts text, detects language, recognizes entities, and detects key phrases.
- Define an index to store the output (raw content, plus pipeline-generated name-value pairs).
- Execute the pipeline to start transformations and analysis, and 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 Storage](#)
- [Postman desktop 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

1. Open this [OneDrive folder](#) and on the top-left corner, click **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. You'll use 7 for this exercise.

1 - Create services

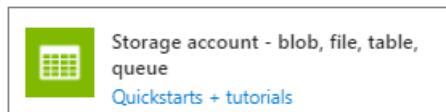
This tutorial uses Azure Cognitive Search for indexing and queries, Cognitive Services on the backend for AI enrichment, and Azure Blob storage to provide the data. This tutorial stays under the free allocation of 20 transactions per indexer per day on Cognitive Services, so the only services you need 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. 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. Click + **Container** to create a container and name it *cog-search-demo*.

8. Select *cog-search-demo* and then click **Upload** to open the folder where you saved the download files. Select all of the non-image files. You should have 7 files. Click **OK** to upload.

<input type="checkbox"/> Name	Date modified	Type	Size
<input checked="" type="checkbox"/> Cognitive Services and Bots (spanish)....	8/22/2019 10:51 PM	Adobe Acrobat D...	3,884 KB
<input checked="" type="checkbox"/> MSFT_cloud_architecture_contoso.pdf	8/22/2019 10:51 PM	Adobe Acrobat D...	1,805 KB
<input checked="" type="checkbox"/> NYSE_LNKD_2015.PDF	8/22/2019 10:51 PM	Adobe Acrobat D...	394 KB
<input checked="" type="checkbox"/> 10-K-FY16.html	8/22/2019 10:51 PM	HTML File	1,835 KB
<input type="checkbox"/> guthrie.jpg	8/22/2019 10:51 PM	JPG File	46 KB
<input type="checkbox"/> redshirt.jpg	8/22/2019 10:51 PM	JPG File	67 KB
<input type="checkbox"/> satyanadellalinux.jpg	8/22/2019 10:51 PM	JPG File	108 KB
<input checked="" type="checkbox"/> Cognitive Seervices and Content Intelli...	8/22/2019 10:51 PM	Microsoft PowerPo...	11,231 KB
<input checked="" type="checkbox"/> MSFT_FY17_10K.docx	8/22/2019 10:51 PM	Microsoft Word D...	675 KB
<input type="checkbox"/> 5074.clip_image002_6FE27E85.png	8/22/2019 10:51 PM	PNG File	472 KB
<input type="checkbox"/> create-search-collect-info.png	8/22/2019 10:51 PM	PNG File	57 KB
<input type="checkbox"/> create-search-service.png	8/22/2019 10:51 PM	PNG File	26 KB
<input type="checkbox"/> create-service-full-portal.png	8/22/2019 10:51 PM	PNG File	67 KB
<input checked="" type="checkbox"/> satyasletter.txt	8/22/2019 10:51 PM	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 *blobstoragewestus* 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:

```
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 Text Analytics 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 7 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

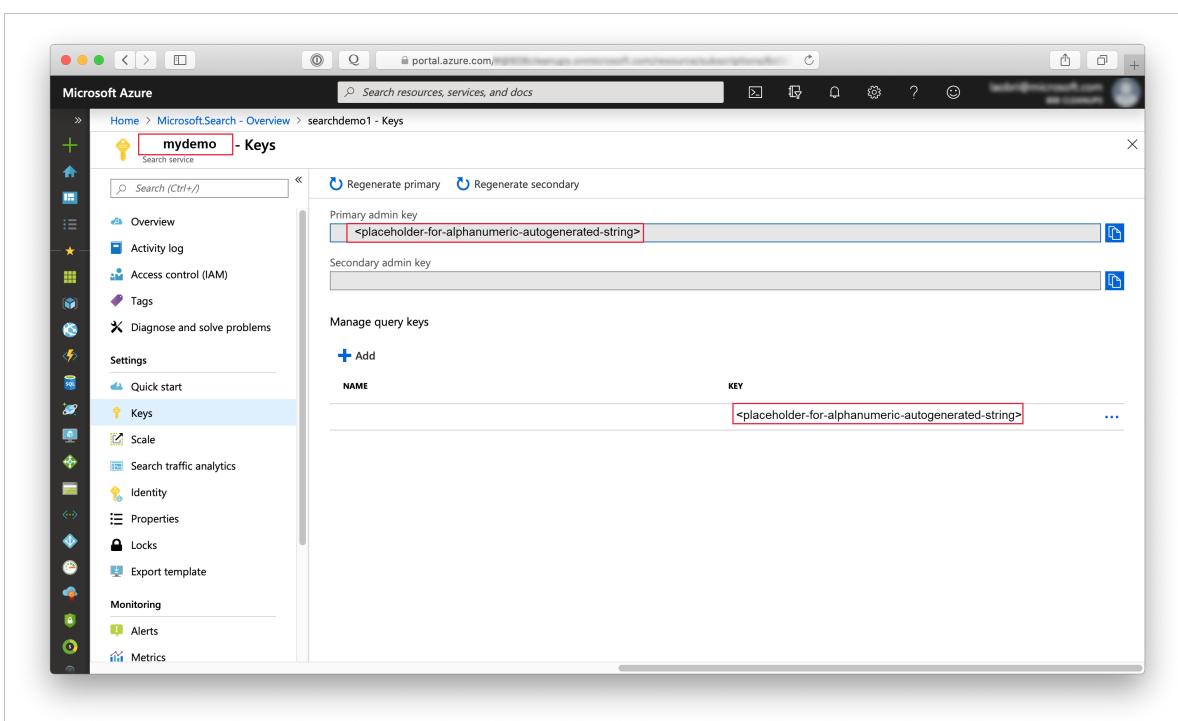
The third component is Azure Cognitive Search, which you can [create in the portal](#). You can use the Free tier to complete this walkthrough.

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 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.

Get the query key as well. It's a best practice to issue query requests with read-only access.



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

Start Postman and set up an HTTP request. If you are unfamiliar with this tool, see [Explore Azure Cognitive Search REST APIs using Postman](#).

The request methods used in this tutorial are **POST**, **PUT**, and **GET**. You'll use the methods to make four API calls to your search service: create a data source, a skillset, an index, and an indexer.

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.

The screenshot shows the Postman interface with a red box highlighting the URL field: `https://mydemo.search.windows.net/indexes?api-version=2020-06-30`. Below the URL, the Headers tab is selected, showing two entries: `Content-Type: application/json` and `api-key: <placeholder-api-key-for-your-service>`. Both entries have a red box around them. At the bottom right, the status bar shows `Status: 200 OK` and `Time: 540 ms`.

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

A [data source object](#) provides the connection string to the Blob container containing the files.

1. Use **POST** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service.

```
https://[YOUR-SERVICE-NAME].search.windows.net/datasources?api-version=2020-06-30
```

2. In request **Body**, copy the following JSON definition, replacing the `connectionString` with the actual connection of your storage account.

Remember to edit the container name as well. We suggested "cog-search-demo" for the container name in an earlier step.

```
{
  "name" : "cog-search-demo-ds",
  "description" : "Demo files to demonstrate cognitive search capabilities.",
  "type" : "azureblob",
  "credentials" :
  { "connectionString" :
    "DefaultEndpointsProtocol=https;AccountName=<YOUR-STORAGE-ACCOUNT>;AccountKey=<YOUR-ACCOUNT-KEY>;"
  },
  "container" : { "name" : "<YOUR-BLOB-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 request construction: `api-version=2020-06-30` should be on the endpoint, `api-key` should be in the Header after `Content-Type`, and its value must be valid for a search service. You might

want to run the JSON document through an online JSON validator to make sure the syntax is correct.

Step 2: Create a skillset

A [skillset object](#) is a set of enrichment steps applied to your content.

1. Use **PUT** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service.

```
https://[YOUR-SERVICE-NAME].search.windows.net/skillsets/cog-search-demo-sd?api-version=2020-06-30
```

2. In request **Body**, copy the JSON definition below. This skillset consists of the following built-in skills.

SKILL	DESCRIPTION
Entity Recognition	Extracts the names of people, organizations, and locations from content in the blob container.
Language Detection	Detects the content's language.
Text Split	Breaks 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	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"`.

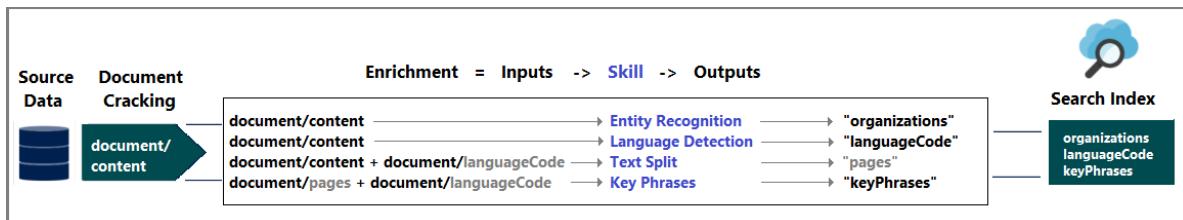
For key phrase extraction, because we use the text splitter skill to break larger files into pages, the context for the key phrase extraction skill is `"document/pages/*"` (for each page in the document) instead of `"/document/content"`.

```

{
  "description": "Extract entities, detect language and extract key-phrases",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
      "categories": [ "Person", "Organization", "Location" ],
      "defaultLanguageCode": "en",
      "inputs": [
        { "name": "text", "source": "/document/content" }
      ],
      "outputs": [
        { "name": "persons", "targetName": "persons" },
        { "name": "organizations", "targetName": "organizations" },
        { "name": "locations", "targetName": "locations" }
      ]
    },
    {
      "@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" }
      ]
    }
  ]
}

```

A graphical representation of the skillset is shown below.



- 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. 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

An [index](#) provides the schema used to create the physical expression of your content in 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 contents and behaviors in Azure Cognitive Search.

1. Use **PUT** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service, to name your index.

```
https://[YOUR-SERVICE-NAME].search.windows.net/indexes/cog-search-demo-idx?api-version=2020-06-30
```

2. In request **Body**, copy the following JSON definition. The `content` field stores the document itself. Additional fields for `languageCode`, `keyPhrases`, and `organizations` represent new information (fields and values) created by the skillset.

```
{
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": true,
      "filterable": false,
      "facetable": false,
      "sortable": true
    },
    {
      "name": "metadata_storage_name",
      "type": "Edm.String",
      "searchable": false,
      "filterable": false,
      "facetable": false,
      "sortable": false
    },
    {
      "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": "persons",
  "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": "locations",
  "type": "Collection(Edm.String)",
  "searchable": true,
  "sortable": false,
  "filterable": true,
  "facetable": true
}
]
```

- Send the request. Postman should return a status code of 201 confirming success.

Step 4: Create and run an indexer

An [Indexer](#) drives 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.

- Use **PUT** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service, to name your indexer.

```
https://[servicename].search.windows.net/indexers/cog-search-demo-idxr?api-version=2020-06-30
```

- In request **Body**, copy the JSON definition below. Notice the field mapping elements; these mappings are important because they define the data flow.

The `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.

The `outputFieldMappings` are for fields created by skills, and thus processed after the skillset has run. The references to `sourceFieldNames` 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.

```

{
  "name": "cog-search-demo-idxr",
  "dataSourceName" : "cog-search-demo-ds",
  "targetIndexName" : "cog-search-demo-idx",
  "skillsetName" : "cog-search-demo-ss",
  "fieldMappings" : [
    {
      "sourceFieldName" : "metadata_storage_path",
      "targetFieldName" : "id",
      "mappingFunction" :
        { "name" : "base64Encode" }
    },
    {
      "sourceFieldName" : "metadata_storage_name",
      "targetFieldName" : "metadata_storage_name",
      "mappingFunction" :
        { "name" : "base64Encode" }
    },
    {
      "sourceFieldName" : "content",
      "targetFieldName" : "content"
    }
  ],
  "outputFieldMappings" :
  [
    {
      "sourceFieldName" : "/document/persons",
      "targetFieldName" : "persons"
    },
    {
      "sourceFieldName" : "/document/organizations",
      "targetFieldName" : "organizations"
    },
    {
      "sourceFieldName" : "/document/locations",
      "targetFieldName" : "locations"
    },
    {
      "sourceFieldName" : "/document/pages/*/keyPhrases/*",
      "targetFieldName" : "keyPhrases"
    },
    {
      "sourceFieldName": "/document/languageCode",
      "targetFieldName": "languageCode"
    }
  ],
  "parameters":
  {
    "maxFailedItems": -1,
    "maxFailedItemsPerBatch": -1,
    "configuration":
    {
      "dataToExtract": "contentAndMetadata",
      "parsingMode": "default",
      "firstLineContainsHeaders": false,
      "delimitedTextDelimiter": ","
    }
  }
}

```

- 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 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": "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

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, send the following request to check the indexer status.

1. Use **GET** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service, to name your indexer.

```
https://[YOUR-SERVICE-NAME].search.windows.net/indexers/cog-search-demo-idxr/status?api-version=2020-06-30
```

2. 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.

NOTE

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.

5 - Search

Now that you've created new fields and information, let's run some queries to understand the value of cognitive search as it relates to a typical search scenario.

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. Use **GET** and the following URL, replacing YOUR-SERVICE-NAME with the actual name of your service, to search for instances of a term or phrase, returning the `content` field and a count of the matching documents.

[https://\[YOUR-SERVICE-NAME\].search.windows.net/indexes/cog-search-demo-idx/docs?search=*&\\$count=true&\\$select=content&api-version=2020-06-30](https://[YOUR-SERVICE-NAME].search.windows.net/indexes/cog-search-demo-idx/docs?search=*&$count=true&$select=content&api-version=2020-06-30)

The results of this query return document contents, which is the same result you would get if used the blob indexer without the cognitive search pipeline. This field is searchable, but unworkable if you want to use facets, filters, or autocomplete.

2. For the second query, return some of the new fields created by the pipeline (persons, organizations, locations, languageCode). We're omitting keyPhrases for brevity, but you should include it if you want to see those values.

```
https://[YOUR-SERVICE-NAME].search.windows.net/indexes/cog-search-demo-idx/docs?  
search=*&$count=true&$select=metadata_storage_name,persons,organizations,locations,languageCode&api-  
version=2020-06-30
```

The fields in the \$select statement contain new information created from the natural language processing capabilities of Cognitive Services. As you might expect, there is some noise in the results and variation across documents, but in many instances, the analytical models produce accurate results.

The following image shows results for Satya Nadella's open letter upon assuming the CEO role at Microsoft.

```
637 {  
638     "@search.score": 1.0,  
639     "metadata_storage_name": "satyasletter.txt",  
640     "languageCode": "en",  
641     "persons": [  
642         "Steve",  
643         "Bill",  
644         "John Thompson",  
645         "Qi Lu",  
646         "Oscar Wilde",  
647         "Satya"  
648     ],  
649     "organizations": [  
650         "Microsoft",  
651         "CEO",  
652         "the Board",  
653         "Nokia devices and services",  
654         "Nokia",  
655         "We",  
656         "Next"  
657     ],  
658     "locations": []  
659 }
```

3. To see how you might take advantage of these fields, add a facet parameter to return an aggregation of

matching documents by location.

```
https://[YOUR-SERVICE-NAME].search.windows.net/indexes/cog-search-demo-idx/docs?  
search=*&facet=locations&api-version=2020-06-30
```

In this example, for each location, there are 2 or 3 matches.

```
21  [
22    {
23      "count": 3,
24      "value": "UNITED STATES"
25    },
26    {
27      "count": 3,
28      "value": "Washington, D.C."
29    },
30    {
31      "count": 2,
32      "value": "Asia"
33    },
34    {
35      "count": 2,
36      "value": "JFK"
37    },
38    {
39      "count": 2,
40      "value": "ONE MICROSOFT WAY"
41    },
42    {
43      "count": 2,
44      "value": "REDMOND, WASHINGTON"
}
```

4. In this final example, apply a filter on the organizations collection, returning two matches for filter criteria based on NASDAQ.

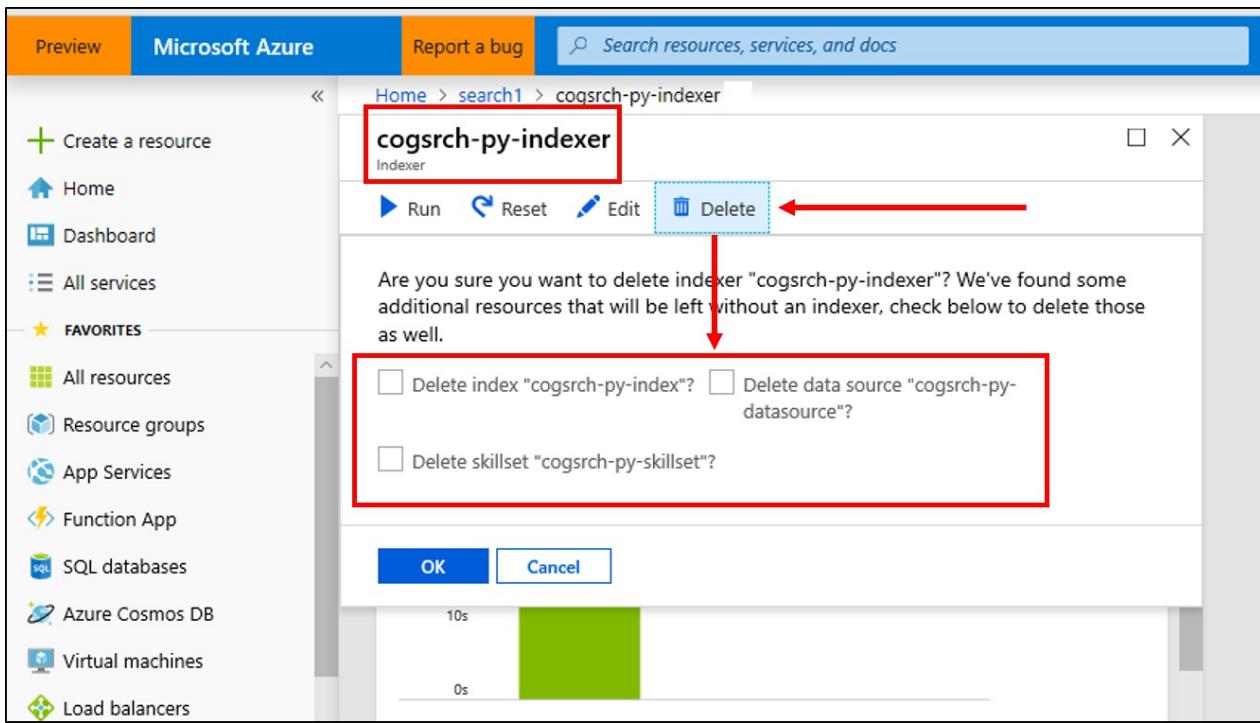
```
https://[YOUR-SERVICE-NAME].search.windows.net/indexes/cog-search-demo-idx/docs?  
search=*&$filter=organizations/any(organizations: organizations eq  
'NASDAQ')&$select=metadata_storage_name,organizations&$count=true&api-version=2020-06-30
```

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

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, data sources, and skillsets. When you delete the indexer, you can optionally, selectively delete the index, skillset, and data source at the same time.



Or use **DELETE** and provide URLs to each object. The following command deletes an indexer.

```
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

10/4/2020 • 15 minutes to read • [Edit Online](#)

If you have unstructured text or images in Azure Blob storage, an [AI enrichment pipeline](#) can extract information and create new content that is useful for full-text search or knowledge mining scenarios. Although a pipeline can process images, this Python tutorial focuses on text, applying language detection and natural language processing to create new fields that you can leverage in queries, facets, and filters.

This tutorial uses Python and the [Search REST APIs](#) to perform the following tasks:

- Start with whole documents (unstructured text) such as PDF, HTML, DOCX, and PPTX in Azure Blob storage.
- Define a pipeline that extracts text, detects language, recognizes entities, and detects key phrases.
- Define an index to store the output (raw content, plus pipeline-generated name-value pairs).
- Execute the pipeline to start transformations and analysis, and 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 Storage](#)
- [Anaconda 3.7](#)
- [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

1. Open this [OneDrive folder](#) and on the top-left corner, click **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. You'll use 7 for this exercise.

1 - Create services

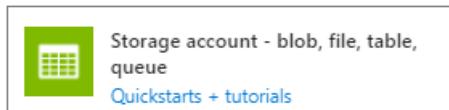
This tutorial uses Azure Cognitive Search for indexing and queries, Cognitive Services on the backend for AI enrichment, and Azure Blob storage to provide the data. This tutorial stays under the free allocation of 20 transactions per indexer per day on Cognitive Services, so the only services you need 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. 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. Click **+ Container** to create a container and name it *cog-search-demo*.

8. Select *cog-search-demo* and then click **Upload** to open the folder where you saved the download files. Select all of the non-image files. You should have 7 files. Click **OK** to upload.

<input type="checkbox"/> Name	Date modified	Type	Size
<input checked="" type="checkbox"/> Cognitive Services and Bots (spanish)....	8/22/2019 10:51 PM	Adobe Acrobat D...	3,884 KB
<input checked="" type="checkbox"/> MSFT_cloud_architecture_contoso.pdf	8/22/2019 10:51 PM	Adobe Acrobat D...	1,805 KB
<input checked="" type="checkbox"/> NYSE_LNKD_2015.PDF	8/22/2019 10:51 PM	Adobe Acrobat D...	394 KB
<input checked="" type="checkbox"/> 10-K-FY16.html	8/22/2019 10:51 PM	HTML File	1,835 KB
guthrie.jpg	8/22/2019 10:51 PM	JPG File	46 KB
redshirt.jpg	8/22/2019 10:51 PM	JPG File	67 KB
satyanadellalinux.jpg	8/22/2019 10:51 PM	JPG File	108 KB
<input checked="" type="checkbox"/> Cognitive Searvices and Content Intelli...	8/22/2019 10:51 PM	Microsoft PowerPo...	11,231 KB
<input checked="" type="checkbox"/> MSFT_FY17_10K.docx	8/22/2019 10:51 PM	Microsoft Word D...	675 KB
5074.clip_image002_6FE27E85.png	8/22/2019 10:51 PM	PNG File	472 KB
create-search-collect-info.png	8/22/2019 10:51 PM	PNG File	57 KB
create-search-service.png	8/22/2019 10:51 PM	PNG File	26 KB
create-service-full-portal.png	8/22/2019 10:51 PM	PNG File	67 KB
<input checked="" type="checkbox"/> satyasletter.txt	8/22/2019 10:51 PM	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 *blobstoragewestus* 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:

```
DefaultEndpointsProtocol=https;AccountName=<storageaccountname>;AccountKey=<your account key>;EndpointSuffix=core.windows.net
```

- 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 Text Analytics 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 7 transactions, you can skip resource provisioning because Azure Cognitive Search can connect to Cognitive Services for 20 free transactions per indexer run. 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](#). You can use the Free tier to complete this walk through.

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 an admin api-key and URL for Azure Cognitive Search

- 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`.

- 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.

Get the query key as well. It's a best practice to issue query requests with read-only access.

NAME	KEY
	<placeholder-for-alphanumeric-autogenerated-string>

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

Create the notebook using the following instructions, or download a finished notebook from [Azure-Search-python-samples repo](#).

Use Anaconda Navigator to launch Jupyter Notebook and create a new Python 3 notebook.

In your notebook, run this script to load the libraries used for working with JSON and formulating HTTP requests.

```
import json
import requests
from pprint import pprint
```

In the same notebook, define the names for the data source, index, indexer, and skillset. Run this script to set up the names for this tutorial.

```
# 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 the following script, replace 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.

```
# 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 walk through 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 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 text for the container. Then, run the script to create a data source named `cogsrch-py-datasource`.

```

# 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. Click **Refresh** to update the page.

TYPE		NAME	TABLE/COLLECTION
	cogsrch-py-datasource		basic-demo-data-pr
	hotels-datasource		hotels
	realestate-us-sample		Listings 5K KingCounty WA

Step 2: Create a skillset

In this step, you will define a set of enrichment steps to apply to your data. You call each enrichment step a *skill*, and the set of enrichment steps a *skillset*. This tutorial uses [built-in cognitive skills](#) for the skillset:

- [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`.

```

# Create a skillset
skillset_payload = {
    "name": skillset_name,
    "description":
        "Extract entities, detect language and extract key-phrases",
    "skills":
    [
        {
            "@odata.type": "#Microsoft.Skills.Text.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)

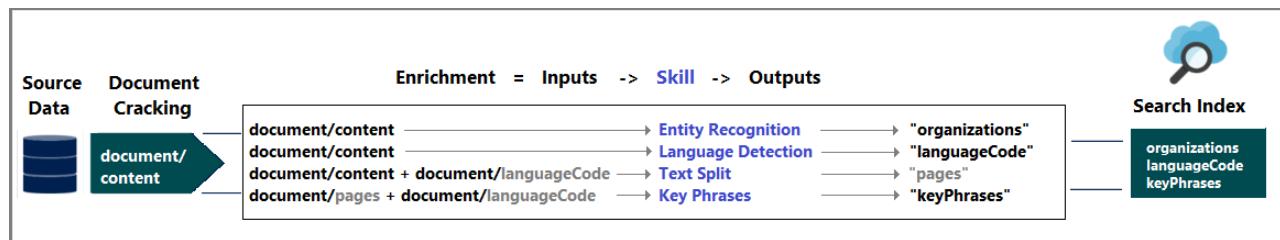
```

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 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 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:	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`.

```

# 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 \(Azure Cognitive Search REST API\)](#).

Step 4: Create and run an indexer

An [Indexer](#) drives 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.

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`.

```
# 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.

```
# 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.

```
#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))

'{"@odata.context":',
'"https://mysearchsvc.search.windows.net/$metadata#Microsoft.Azure.Search.V2019_05_06.IndexerExecutionInfo",\n',
'"name": "cogrchr-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\": \"\n',
'  \"0001-01-01T00:00:00Z\\\",\\r\\n  \"lastAttemptedEnumerationStartTime\": \"\n',
'  \"0001-01-01T00:00:00Z\\\",\\r\\n  \"nameHighWaterMark\": null\\r\\n}\",\n',
'  \"finalTrackingState\": \n',
'  \"\"\\\"LastFullEnumerationStartTime\\\":\\\"2019-06-13T15:14:44.0784124+00:00\\\",\\\"LastAttemptedEnumerationStartTime\\\":\\\"201\n',
'  9-06-13T15:14:44.0784124+00:00\\\",\\\"NameHighWaterMark\\\":null}\",\n',
'  \"errors\": []\n'}'
```

Warnings are common with some source file and skill combinations and do not always indicate a problem. Many warnings are benign. For example, if you index a JPEG file that does not have text, you will see the warning in this screenshot.

```
#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"
},
{
    "key": '',
    "url": "https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/NYSE_LNKD_2015.PDF",
    "message": "Truncated extracted text to 65536 characters.\n"
},
{
    "key": '',
    "url": "https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/create-service-full-portal.png",
    "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\r\n'
},
{
    "key": '',
    "url": "https://mysearchsvc.blob.core.windows.net/basic-demo-data-pr/create-search-service.png",
    "message": "Skill #1: Input text is missing. No entities will be "
'returned\r\nSkill #2: Input text is missing. No language will be '

```

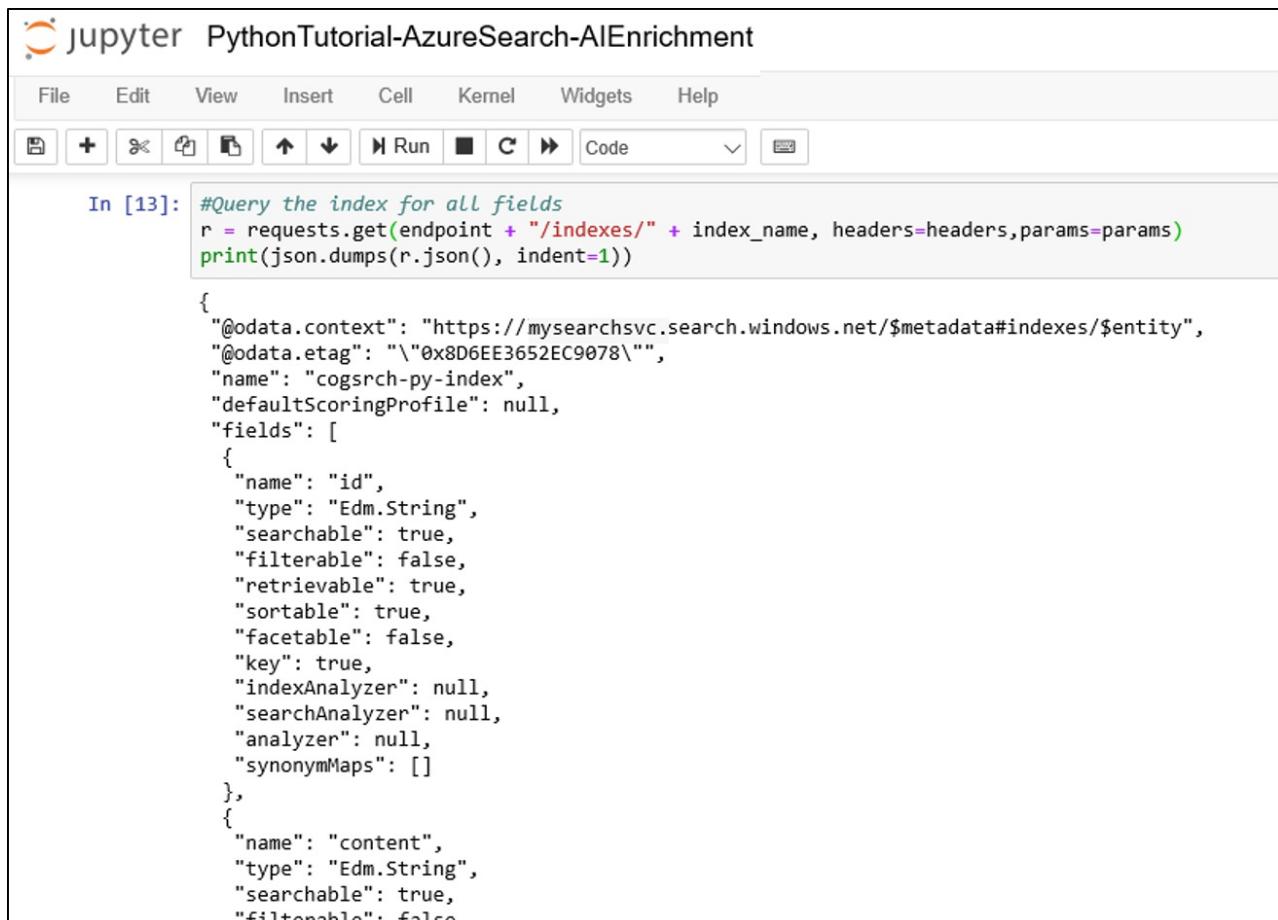
5 - Search

After indexing is finished, run queries that return the contents of individual fields. By default, Azure Cognitive Search returns the top 50 results. The sample data is small so the default works fine. However, when working with larger data sets, you might need to include parameters in the query string to return more results. For instructions, see [How to page results in Azure Cognitive Search](#).

As a verification step, get the index definition showing all of the fields.

```
# 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))
```

The results should look similar to the following example. The screenshot only shows a part of the response.



The screenshot shows a Jupyter Notebook interface with the title "jupyter PythonTutorial-AzureSearch-AIEnrichment". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and various icons for file operations. In the code cell (In [13]), the following Python code is displayed:

```
In [13]: #Query the index for all fields
r = requests.get(endpoint + "/indexes/" + index_name, headers=headers, params=params)
print(json.dumps(r.json(), indent=1))
```

The output of the code is a JSON object representing the index definition. The JSON structure includes the context URL, ETag, name, default scoring profile, and two fields: "id" and "content". The "id" field is defined with type "Edm.String", searchable true, filterable false, retrievable true, sortable true, facetable false, key true, and no analyzers or synonym maps. The "content" field is also defined with type "Edm.String", searchable true, filterable false.

```
{
    "@odata.context": "https://mysearchsvc.search.windows.net/$metadata#indexes/$entity",
    "@odata.etag": "\"0x8D6EE3652EC9078\"",
    "name": "cogsrch-py-index",
    "defaultScoringProfile": null,
    "fields": [
        {
            "name": "id",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false,
            "retrievable": true,
            "sortable": true,
            "facetable": false,
            "key": true,
            "indexAnalyzer": null,
            "searchAnalyzer": null,
            "analyzer": null,
            "synonymMaps": []
        },
        {
            "name": "content",
            "type": "Edm.String",
            "searchable": true,
            "filterable": false
        }
    ]
}
```

The output is the index schema, with the name, type, and attributes of each field.

Submit a second query for `"*"` to return all contents of a single field, such as `organizations`.

```
# 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))
```

The results should look similar to the following example. The screenshot only shows a part of the response.

The screenshot shows a Jupyter Notebook interface with the title "jupyter PythonTutorial-AzureSearch-AIEnrichment". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, and Python 3. The toolbar has icons for file operations like Open, Save, and Run. In the code cell, the following Python code is shown:

```
In [14]: #Query the index to return the contents of organizations
#Note: Index creation may take time. If this step returns no data, wait a few minutes
#       and then try again
r = requests.get(endpoint + "/indexes/" + index_name + "/docs?&search=*&$select=organizations", headers=headers, params=params)
pprint(r.json())
```

The output cell displays the JSON response, which includes the context URL and a list of organization names. The list starts with "Microsoft" and includes various other Microsoft-related terms like "Cloud", "Contoso", "Windows Server AD", "Management", "Azure IaaS", "Enterprise Architects", "Cloud Networking", and "Cloud Security".

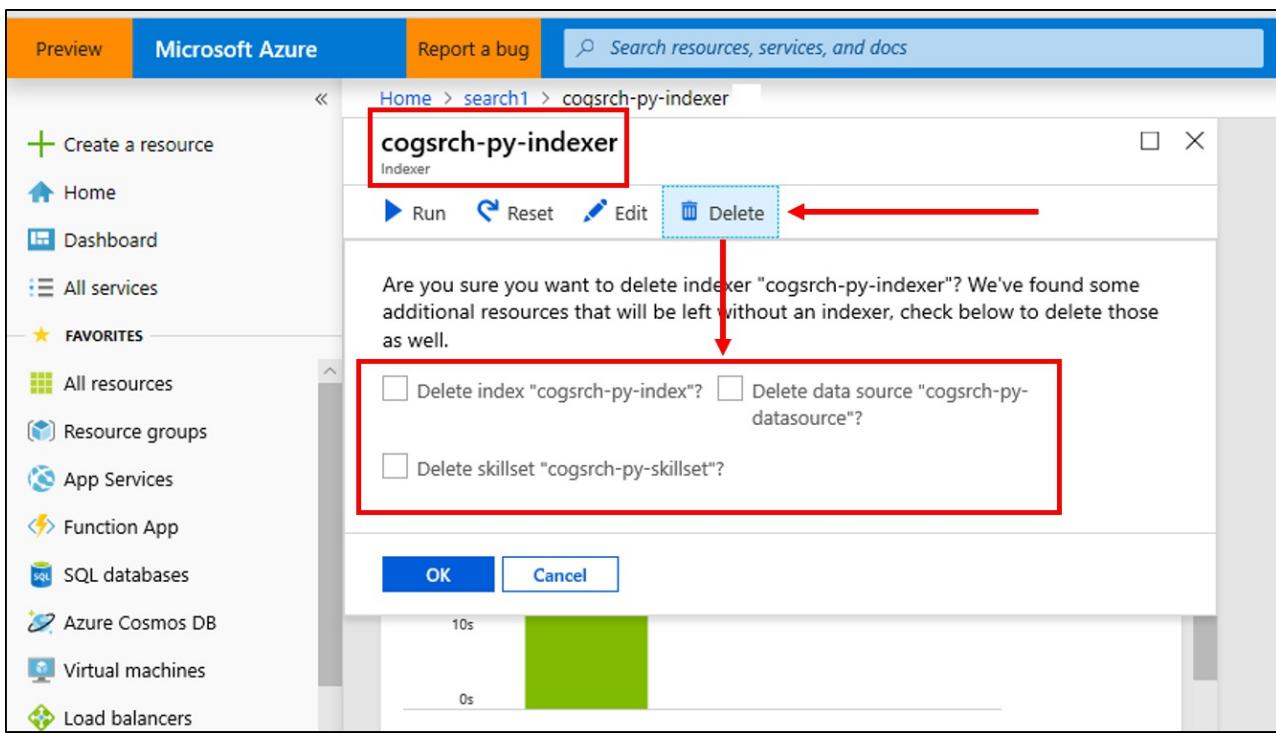
Repeat for additional fields: `content`, `languageCode`, `keyPhrases`, and `organizations` in this exercise. You can return multiple fields via `$select` using a comma-delimited list.

You can use GET or POST, depending on query string complexity and length. For more information, see [Query using the REST API](#).

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, 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 shows how to delete a skillset.

```
# 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 is 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: Diagnose, repair, and commit changes to your skillset

10/4/2020 • 11 minutes to read • [Edit Online](#)

In this article, you will use the Azure portal to access Debug sessions to repair issues with the provided skillset. The skillset has some errors which need to be addressed. This tutorial will take you through a debug session to identify and resolve issues with skill inputs and outputs.

IMPORTANT

Debug sessions is a preview feature provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

Prerequisites

- An Azure subscription. Create a [free account](#) or use your current subscription
- An Azure Cognitive Search service instance
- An Azure Storage account
- [Postman desktop app](#)

Create services and load data

This tutorial uses Azure Cognitive Search and Azure Storage services.

- [Download sample data](#) consisting of 19 files.
- [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.

- Open the storage services pages and create a container. Best practice is to specify the access level "private". Name your container `clinicaltrialdataset`.
- In container, click **Upload** to upload the sample files you downloaded and unzipped in the first step.
- [Create an Azure Cognitive Search service](#) or [find an existing service](#). You can use a free service for this quickstart.

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.

The screenshot shows the Azure portal interface. At the top, there's a navigation bar with 'Dashboard > mydemo'. Below it, the main content area shows a 'Search service' named 'mydemo'. A red box highlights the 'Overview' tab in the left sidebar. In the top right, there are several buttons: 'Add index', 'Import data', 'Search explorer', 'Refresh', 'Delete', and 'Move'. A red circle labeled '1' is positioned over the 'Url' field, which contains the value 'https://mydemo.search.windows.net'. Another red box highlights this URL field. In the bottom right corner of the main pane, there's a 'Pricing tier' section. The bottom part of the screenshot shows the 'Keys' page for the service. The left sidebar has tabs for 'Overview', 'Activity log', 'Access control (IAM)', 'Settings', 'Quick start', and 'Keys'. A red box highlights the 'Keys' tab. The main content area shows two fields: 'Primary admin key' and 'Secondary admin key', both containing placeholder text '<placeholder-for-alphanumeric-autogenerated-string>'. A red circle labeled '2' is positioned over the 'Keys' tab in the sidebar.

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 search service's data source, skillset, index, and indexer.

1. If you do not have Postman, you can [download the Postman desktop app here](#).
2. [Download the Debug Sessions Postman collection](#)
3. Start Postman
4. Under **Files** > **New**, select the collection to import.
5. After the collection is imported, expand the actions list (...).
6. Click **Edit**.
7. Enter the name of your searchService (for example, if the endpoint is `https://mydemo.search.windows.net`, then the service name is "`mydemo`").
8. Enter the apiKey with either the primary or secondary key of your search service.
9. Enter the storageConnectionString from the keys page of your Azure Storage account.
10. Enter the containerName for the container you created in the storage account.

EDIT COLLECTION

Name

Description Authorization Pre-request Scripts Tests **Variables** •

These variables are specific to this collection and its requests. [Learn more about collection variables.](#)

	VARIABLE	INITIAL VALUE i	CURRENT VALUE i	...	Persist All	Reset All
<input checked="" type="checkbox"/>	searchService					
<input checked="" type="checkbox"/>	apiKey					
<input checked="" type="checkbox"/>	storageConnectionString					
<input checked="" type="checkbox"/>	containerName					
	Add a new variable					

i Use variables to reuse values in different places. The current value is used while sending a request and is never synced to Postman's servers. The initial value is auto-updated to reflect the current value. [Change this](#) behaviour from Settings. X

[Learn more about variable values](#)

Cancel Update

The collection contains four different REST calls that are used to complete this section.

The first call creates the data source. `clinical-trials-ds`. The second call creates the skillset, `clinical-trials-ss`. The third call creates the index, `clinical-trials`. The fourth and final call creates the indexer, `clinical-trials-idxr`. After all of the calls in the collection have been completed, close Postman and return to the Azure portal.

POST CreateDataSource

POST https://{{searchService}}.search.windows.net/datasources?api-version=2019-05-06-Preview

Headers (1)

KEY	VALUE	DESCRIPTION
apiKey	{{apiKey}}	

Check the results

The skillset contains a few, common errors. In this section, running an empty query to return all documents will display multiple errors. In subsequent steps, the issues will be resolved using a debug session.

1. Go to your search service in the Azure portal.
2. Select the **Indexes** tab.
3. Select the `clinical-trials` index
4. Click **Search** to run an empty query.

After the search has finished, two fields with no data listed; "organizations" and "locations" are listed in the window. Follow the steps to discover all of the issues produced by the skillset.

1. Return to the search service Overview page.
2. Select the **Indexers** tab.
3. Click `clinical-trials-idxr` and select the warnings notification.

There are many issues with projection output field mappings and on page three there are warnings because one or more skill inputs are invalid.

Return to the search service overview screen.

Start your debug session

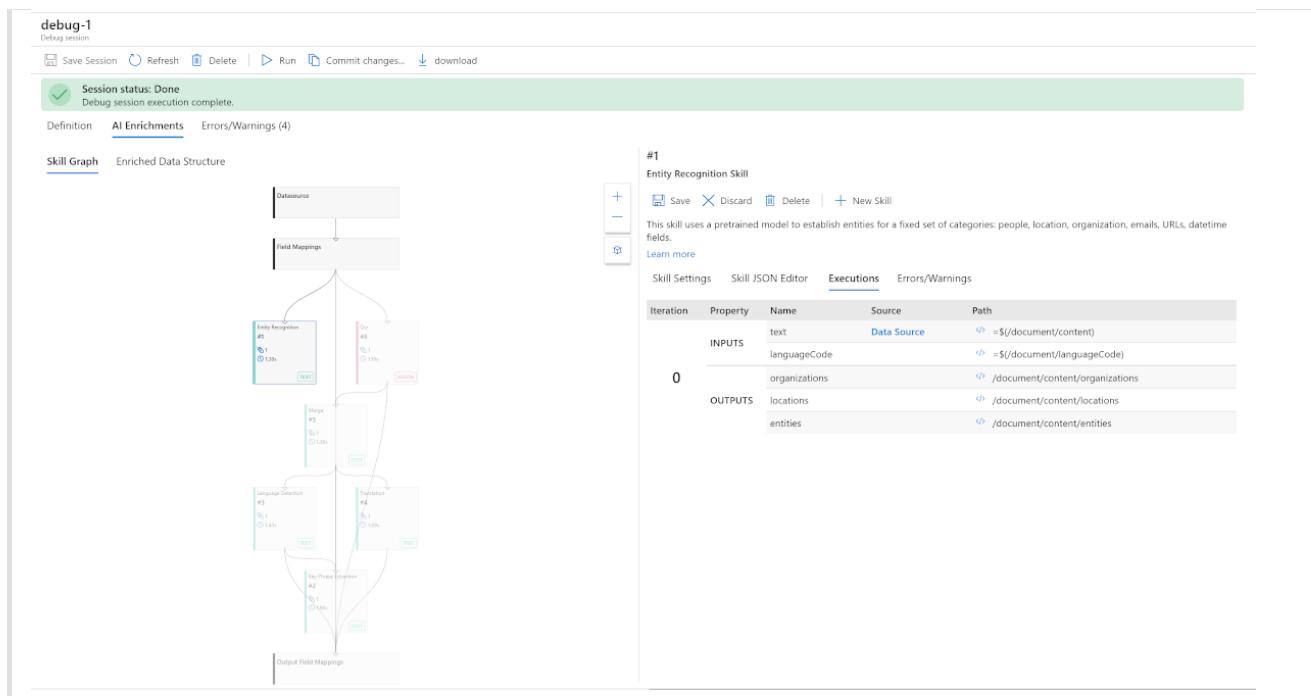
The screenshot shows the 'new-debug-session' configuration page in the Azure portal. The 'Definition' tab is selected. The form includes fields for:

- Debug session name ***: new-debug-session
- Description**: (optional)
- Storage connection string * ⓘ**: Choose an existing connection
- Indexer * ⓘ**: clinical-trials-idxr
Skillset: clinical-trials-ss Data Source: clinical-trials-ds
- Datasource**: clinical-trials-ds
- Document to debug ⓘ**: Select first document

1. Click on the Debug sessions (preview) tab.
2. Select +NewDebugSession
3. Give the session a name.
4. Connect the session to your storage account.
5. 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.
7. Save the session. Saving the session will kick off the AI enrichment pipeline as defined by the skillset.

IMPORTANT

A debug session only works with a single document. A specific document in the data set can be > selected or the session will default to the first document.



When the debug session has finished executing, 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 from top to bottom. Skills that are side by side in the graph are executed in parallel. Color coding of skills in the graph indicate the types of skills that are being executed in the skillset. In the example, the green skills are text and the red skill is vision. Clicking on an individual skill in the graph will display the details of that instance of the skill in the right pane of the session window. The skill settings, a JSON editor, execution details, and errors/warnings are all available for review and editing.
- The Enriched Data Structure details the nodes in the enrichment tree generated by the skills from the source document's contents.

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.

Fix missing skill input value

In the Errors/Warnings tab, there is an error for an operation labeled `Enrichment.NerSkillv2.#1`. The detail of this error explains that there is a problem with a skill input value '`/document/languageCode`'.

1. Return to the AI Enrichments tab.
2. Click on the **Skill Graph**.
3. Click on the skill labeled `#1` to display its details in the right pane.
4. Locate the input for "languageCode".
5. Select the `</>` symbol at the beginning of the line and open the Expression Evaluator.
6. Click the **Evaluate** button to confirm that this expression is resulting in an error. It will confirm that the "languageCode" property is not a valid input.

The screenshot shows the AI Skills interface with the 'debug-1' session open. The left pane displays the 'Skill Graph' where nodes like 'Datasource', 'Field Mappings', 'Entity Recognition #1', 'Language Detector #2', and 'Output Field Mappings' are interconnected. The right pane shows the 'Expression evaluator' for skill #1, with the context set to '/document/content'. An error message states 'The expression did not resolve to a valid input path.' Below the message is a code editor containing JSON-like code for defining inputs and outputs.

```

    {
      "inputs": [
        {
          "path": "/document/languageCode",
          "source": "#1"
        }
      ],
      "outputs": [
        {
          "path": "/document/entities",
          "name": "entities",
          "targetName": "entities"
        },
        {
          "path": "/document/locations",
          "name": "locations",
          "targetName": "locations"
        },
        {
          "path": "/document/organizations",
          "name": "organizations",
          "targetName": "organizations"
        }
      ]
    }
  
```

There are two ways to research this error in the session. 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. Click the **Executions** tab.
2. Look at the INPUTS and find "languageCode". There is no source for this input listed.
3. Switch the left pane to display the Enriched Data Structure. There is no mapped path corresponding to "languageCode".

The screenshot shows the AI Skills interface with the 'debug-1' session open. The left pane displays the 'Enriched Data Structure' with a tree view of paths like 'document', 'languageName', 'merged_content', etc. The right pane shows the 'Skill Settings' for skill #1, specifically the 'Entity Recognition Skill' settings. The 'Skill JSON Editor' tab is active, showing the skill's configuration. The 'Executions' tab is also visible, showing a table of inputs and outputs with their sources.

Iteration	Property	Name	Source	Path
0	INPUTS	text	Data Source	=\$/document/content
		languageCode		=\$#/document/languageCode
0	OUTPUTS	organizations		\$/document/content/organizations
		locations		\$/document/content/locations
		entities		\$/document/content/entities

There is a mapped path for "language." So, there is a typo in the skill settings. To fix this the expression in the #1 skill with the expression '/document/language' will need to be updated.

1. Open the Expression Evaluator </> for the path "language."
2. Copy the expression. Close the window.
3. Go to the Skill Settings for the #1 skill and open the Expression Evaluator </> for the input "languageCode."
4. Paste the new value, '/document/language' into the Expression box and click **Evaluate**.

5. It should display the correct input "en". Click Apply to update the expression.
6. Click **Save** in the right, skill details pane.
7. Click **Run** in the session's window menu. This will kick off another execution of the skillset using the document.

Once the debug session execution completes, click the Errors/Warnings tab and it will show that the error labeled "Enrichment.NerSkillV2.#1" is gone. However, there are still two warnings that the service could not map output fields for organizations and locations to the search index. There are missing values: '/document/merged_content/organizations' and '/document/merged_content/locations'.

Fix missing skill output values

The screenshot shows the Data Pipeline interface with the 'Errors/Warnings' tab selected. There are two warning entries:

- Projection.SearchIndex.OutputFieldMapping.locations: Could not map output field 'locations' to search index. Check your indexer's 'outputFieldMappings' prop... Missing value '/document/merged_content/locations'.
- Projection.SearchIndex.OutputFieldMapping.organizations: Could not map output field 'organizations' to search index. Check your indexer's 'outputFieldMappings' ... Missing value '/document/merged_content/organizations'.

There are missing output values from a skill. To identify the skill with the error go to the Enriched Data Structure, find the value name and look at its Originating Source. In the case of the missing organizations and locations values, they are outputs from skill #1. Opening the Expression Evaluator </> for each path, will display the expressions listed as '/document/content/organizations' and '/document/content/locations', respectively.

The screenshot shows the Data Pipeline interface with the 'Skill Graph' and 'Enriched Data Structure' tabs selected. The 'Enriched Data Structure' tab displays a table of paths and their corresponding output and originating source:

Path	Output	Originating Source
document	{"normalized_images": [{"type": "file", "url": "@trim..."}]}	#5
merged_content	"\n \n\n[image: image0.jpg] Study of BMN 110 in ..."	#2
keyphrases	["Study of BMN", "Syndrome", "Pediatric Patients", "Y..."]	#4
translated_text	"\n[image: image0.jpg] Study of BMN 110 in Pedia..."	#3
language	"en"	#1
content		#1
entities		#1
locations		#1
organizations		#1

A modal window titled 'Expression evaluator' is open, showing the context '/document' and the expression '/document/content/organizations'. The value field is empty, indicated by a small icon.

The output for these entities is empty and it should not be empty. What are the inputs producing this result?

1. Go to **Skill Graph** and select skill #1.
2. Select **Executions** tab in the right skill details pane.
3. Open the Expression Evaluator </> for the INPUT "text."

The screenshot shows the MuleSoft Anypoint Studio interface. On the left, there's a sidebar with icons for saving, discarding, deleting, and creating new skills. The main area displays the details for skill #1, titled "Entity Recognition Skill". It states that the skill uses a pretrained model to establish entities for categories like people, location, organization, email, and phone number. A "Learn more" link is available. Below this, tabs for "Skill Settings", "Skill JSON Editor", "Executions", and "Errors/Warnings" are present, with "Executions" being the active tab. The "Executions" pane shows a table with one row for an input named "text". The "Source" column shows the expression `=${/document/content}`. The "Path" column shows a hierarchy starting with `=${/document/language}`, followed by `/document/content/organizations`, `/document/content/locations`, and `/document/content/entities`. To the left of the table, there's an "Expression evaluator" section with "Context" set to `/document` and an "Expression" field containing `=${/document/content}`. A "Evaluate" button is next to the expression field. Below this, a "Value" section shows the result of the evaluation: `"\n \n\n[image: image0.jpg]\n\n"`. There's also a "Wrap Columns" checkbox.

The displayed result for this input doesn't look like a text input. It looks like an image that is surrounded by new lines. The lack of text means that no entities can be identified. Looking at the hierarchy of the skillset displays the content is first processed by the #6 (OCR) skill and then passed to the #5 (Merge) skill.

1. Select the #5 (Merge) skill in the **Skill Graph**.
2. Select the **Executions** tab in the right skill details pane and open the Expression Evaluator `</>` for the OUTPUTS "mergedText".

The screenshot shows the MLOps Platform interface. On the left, there's a sidebar with a yellow header and a green footer. The main area has a light blue header. Below the header, there's a section for 'Merge Skill' with options to 'Save', 'Discard', 'Delete', and 'New Skill'. It also says 'Consolidates text from a collection of fields into a single field.' and has a 'Learn more' link. Below this are tabs for 'Skill Settings', 'Skill JSON Editor', 'Executions' (which is selected), and 'Errors/Warnings'. The 'Executions' tab shows a table with one row for iteration 0. The row has columns for 'Iteration', 'Property', 'Name', and 'Source'. The 'text' property is set to 'Data Source'. The 'itemsToInsert' property is set to '#6 Data Source' and the 'offsets' property is set to 'Data Source'. Under 'OUTPUTS', the 'mergedText' output is shown with the source '/document/merged_content'. To the right of the table is an 'Expression Evaluator' window. The 'Context' is set to '/document'. The 'Expression' is set to '/document/merged_content'. The 'Value' section contains a red error message: "\n \n\n[image: image0.jpg] Study of BMN 110 in Pediatric Patients < 5 Years of Age With Mucopolysaccharidosis IVA (Morquio A Syndrome) This open-label Phase 2 study will evaluate the safety and efficacy of weekly 2.0 mg/kg/wk infusions of BMN 110 in pediatric patients, less than 5 years of age at the time of administration of the first dose of study drug, diagnosed with MPS IVA (Morquio A Syndrome) for up to 208 weeks. \n\n\n". There is a 'Close' button at the bottom of the evaluator window.

Here the text is paired with the image. Looking at the expression '/document/merged_content' the error in the "organizations" and "locations" paths for the #1 skill is visible. Instead of using '/document/content' it should use '/document/merged_content' for the "text" inputs.

1. Copy the expression for the "mergedText" output and close the Expression Evaluator window.
2. Select skill #1 in the **Skill Graph**.
3. Select the **Skill Settings** tab in the right skill details pane.
4. Open the Expression Evaluator </> for the "text" input.
5. Paste the new expression into the box. Click **Evaluate**.
6. The correct input with the added text should be displayed. Click **Apply** to update the Skill Settings.
7. Click **Save** in the right, skill details pane.
8. Click **Run** in the sessions window menu. This will kick off another execution of the skillset using the document.

After the indexer has finished running, the errors are still there. Go back to skill #1 and investigate. The input to the skill was corrected to 'merged_content' from 'content'. What are the outputs for these entities in the skill?

1. Select the **AI Enrichments** tab.
2. Select **Skill Graph** and click on skill #1.
3. Navigate the **Skill Settings** to find "outputs."
4. Open the Expression Evaluator </> for the "organizations" entity.

The screenshot shows the Alteryx Expression evaluator dialog box. The context is set to '/document/content'. The expression is '/document/content/organizations'. The value is a JSON array: [1, "BMN", 3]. Below the dialog, a tooltip displays the following JSON structure:

```
↳ — {name : "organizations", targetName : "organizations" }  
↳ — {name : "locations", targetName : "locations" }  
↳ — {name : "entities", targetName : "entities" }  
+
```

Evaluating the result of the expression gives the correct result. The skill is working to identify the correct value for the entity, "organizations." However, the output mapping in the entity's path is still throwing an error. In comparing the output path in the skill to the output path in the error, the skill that is parenting the outputs, organizations and locations under the /document/content node. While the output field mapping is expecting the results to be parented under the /document/merged_content node. In the previous step, the input changed from '/document/content' to '/document/merged_content'. The context in the skill settings needs to be changed in order to ensure the output is generated with the right context.

1. Select the **AI Enrichments** tab.
2. Select **Skill Graph** and click on skill #1.
3. Navigate the **Skill Settings** to find "context."
4. Double-click the setting for "context" and edit it to read '/document/merged_content'.
5. Click **Save** in the right, skill details pane.
6. Click **Run** in the sessions window menu. This will kick off another execution of the skillset using the document.

The screenshot shows the 'Skill Settings' tab selected in the Entity Recognition Skill configuration interface. The skill is named '#1'. The 'context' field is set to '/document/merged_content'. The 'categories' field contains three entries: 'Person', 'Quantity', and 'Organization'. The 'Skill JSON Editor' tab is also visible.

```
{  
  name : "#1",  
  description : null,  
  context : "/document/merged_content",  
  categories : [  
    "Person",  
    "Quantity",  
    "Organization"]}
```

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 so changes made would not affect the production system. Now that you have finished debugging your skillset, the fixes can be committed (overwrite the original skillset) to the production system. If you want to continue to make changes to the skillset without impacting the production system, the debug session can be saved and reopened later.

1. Click **Commit changes** in the main Debug sessions menu.
2. Click **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. Click **Reset**.
6. Click **Run**. Click **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. Exit **Indexer** and select the **Index** tab.
2. Open the 'clinical-trials' index and in the Search explorer tab, click **Search**.
3. The result window should show that organizations and locations entities 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

[Learn more about skillsets](#) [Learn more about incremental enrichment and caching](#)

Tutorial: Create a custom analyzer for phone numbers

10/4/2020 • 12 minutes to read • [Edit Online](#)

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 desktop 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 [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.

The screenshot shows the Postman interface with a red box highlighting the 'Headers' tab and the two header entries: 'Content-Type: application/json' and 'api-key: <placeholder-api-key-for-your-service>'. The status bar at the bottom right indicates 'Status: 200 OK' and 'Time: 540 ms'.

If you're unfamiliar with Postman, see [Explore Azure Cognitive Search REST APIs using Postman](#).

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:

```
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:

```

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`:

```

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:

```
{  
    "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

```
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**.

```
{  
    "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

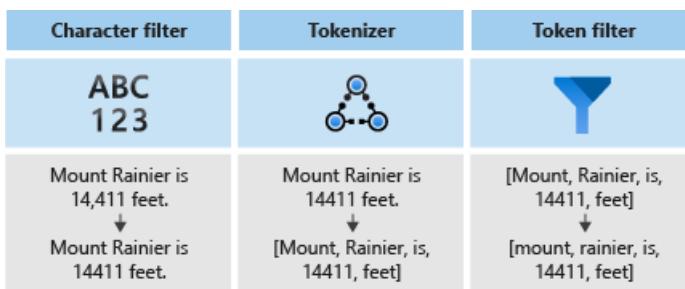
An [analyzer](#) is a component of the [full text search engine](#) responsible for processing text in query strings and indexed documents. Different analyzers manipulate text in different ways depending on the scenario. For this scenario, we need to build an analyzer tailored to phone numbers.

Analyzers consist of three components:

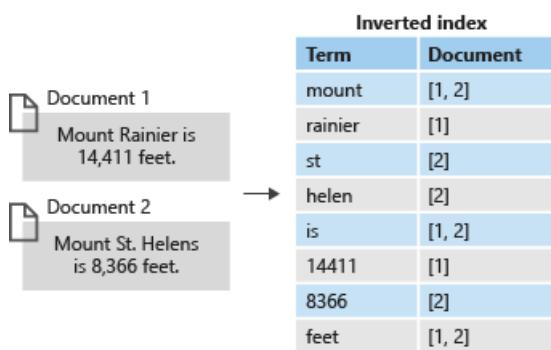
- [Character filters](#) that remove or replace individual characters from the input text.

- 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:

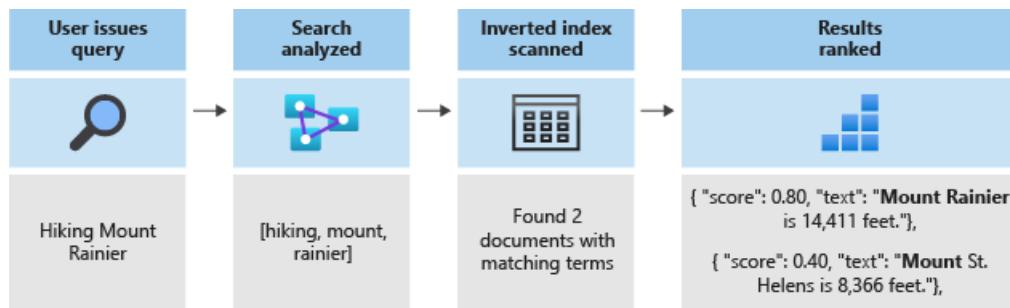


These tokens are then stored in an inverted index, which allows for fast, full-text searches. An inverted index enables full-text search by mapping all unique terms extracted during lexical analysis to the documents in which they occur. You can see an example in the diagram below:



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 [similarity 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:

```
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:

```
{
    "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.

```
{
    "text": "4255550100",
    "analyzer": "standard.lucene"
}
```

```
{
    "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.

```
"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
<code>(321) 555-0199</code>	<code>3215550199</code>
<code>321.555.0199</code>	<code>3215550199</code>

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
<code>The dog swims.</code>	<code>[The dog swims.]</code>
<code>3215550199</code>	<code>[3215550199]</code>

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.

```
"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.

```
"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]

INPUT	OUTPUT
(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:

```
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:

```
{
  "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`.

```
{  
    "@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`.

```
{  
    "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.

```

"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:

```

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

10/4/2020 • 7 minutes to read • [Edit Online](#)

Leverage 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.

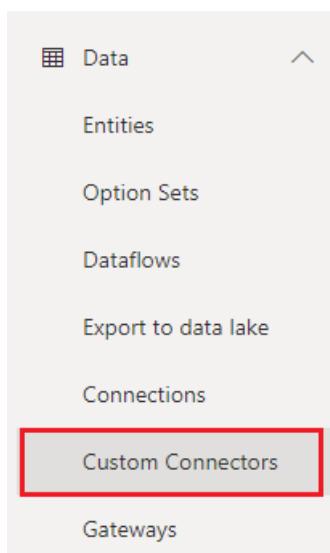
Prerequisites

- [Power Apps account](#)
- [Hotels-sample index](#)
- [Query API key](#)

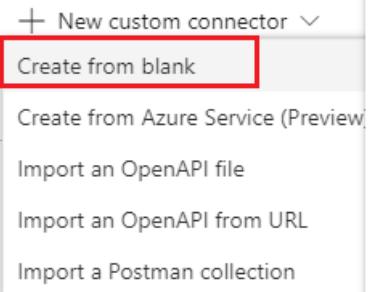
1 - Create a custom connector

A connector in Power Apps is a data source connection. In this step, you'll create a custom connector to connect to a search index in the cloud.

1. [Sign in](#) to Power Apps.
2. On the left, expand Data > 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 click **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, you will need to enter your search service URL (such as
`<your servicename>.search.windows.net`)
- For Base URL, simply enter "/"

General information



Upload connector icon
Supported file formats are PNG and JPG. (< 1MB)

↑ Upload

Icon background color
#007ee5

Description
A connector to Azure Cognitive Search

Connect via on-premises data gateway [Learn more](#)

Scheme *
 HTTPS HTTP

Host *
mydemo.search.windows.net

Base URL
/

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 below.

Authentication type

Choose what authentication is implemented by your API *

API Key

 Edit

API Key

Users will be required to provide the API Key when creating a connection

Parameter label *

api-key

Parameter name *

api-key

Parameter location *

Header

7. In the Definitions Page, select + New Action to create an action that will query the index. Enter the value "Query" for the summary and the name of the operation ID. Enter a description like "*Queries the search index*".

General

Summary [Learn more](#)

Query

Description [Learn more](#)

Queries an Azure Cognitive Search index

Operation ID *

This is the unique string used to identify the operation.

Query

Visibility [Learn more](#)

none advanced internal important

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`.

Power Apps will use the syntax to extract parameters from the query. Notice we explicitly defined the search field.

Import from sample

X

Verb *

- GET DELETE POST PUT HEAD
 OPTIONS PATCH

URL *

`https://mydemo.search.windows.net/indexes/hotels-sample-index/docs`

This is the request URL.

Headers

`Content-Type application/json`

These are custom headers that are part of the request.

Import

Close

9. Click **Import** to auto-fill the Request. Complete setting the parameter metadata by clicking the ... symbol next to each of the parameters. Click **Back** to return to the Request page after each parameter update.

Request + Import from sample

Verb *
The verb describes the operations available on a single path.
GET

URL *
This is the request URL.
`https://mydemo.search.windows.net/indexes/hotels-sample-index/docs`

Path
Path is used together with Path Templating, where the parameter value is actually part of the operation's URL.

Query
Query parameters are appended to the URL. For example, in /items?id=####, the query parameter is id.
`search` ... `$select` ... `api-version` ...

Headers
These are custom headers that are part of the request.
`Content-Type` ...

Body
The body is the payload that's appended to the HTTP request. There can only be one body parameter.

10. For `search`: Set `*` as the default value, set required as *False* and set visibility to *none*.

Parameter

Name *

search

Description [Learn more](#)

Summary [Learn more](#)

Default value

*

Is required?

Yes No

Visibility [Learn more](#)

none advanced internal important

11. For `select`: Set `HotelName,Description,Address/City` as the **default value**, set **required** to *False*, and set **visibility** to *none*.

Parameter

Name *

\$select

Description [Learn more](#)

Summary [Learn more](#)

Default value

HotelName,Description,Address/City

Is required?

Yes No

Visibility [Learn more](#)

none advanced internal important

12. For `api-version`: Set `2020-06-30` as the **default value**, set **required** to *True*, and set **visibility** as *internal*.

Parameter

Name *

Description [Learn more](#)

Summary [Learn more](#)

Default value

Is required?

Yes No

Visibility [Learn more](#)

none advanced internal important

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:

```
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. Return to the **3. Request** step and scroll down to the Response section. Click "**Add default response**". This is critical because it will help 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.

```
{
    "@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.

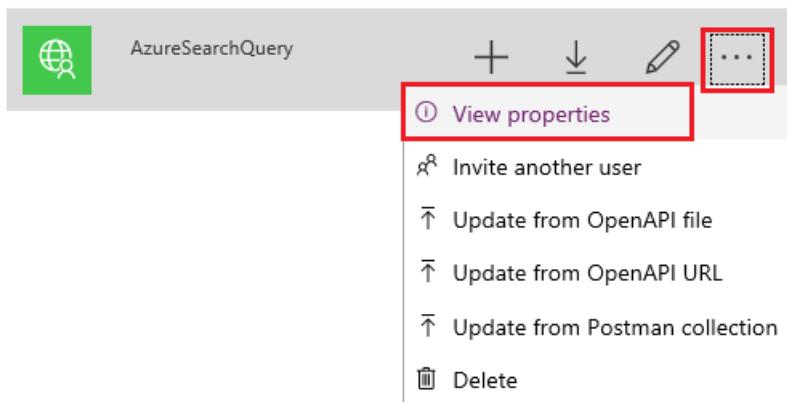
17. Click **Create connector** on the top right.

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 additional updates, you can test from within the wizard.

You will 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, click **Custom Connectors**.
2. Search for the connector by name (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 **4. Test** to open the test page.
6. In Test Operation, click **+ 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, click the **Test operation** button. If you are successful you should see a 200 status, and in the body of the response you should see JSON that describes the search results.

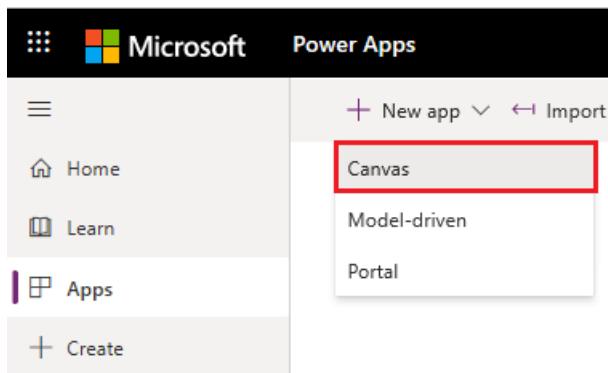
The screenshot shows the "Test operation" results in the Azure portal. It has tabs for "Request" and "Response", with "Response" selected. The "Status" is (200). The "Headers" section shows a JSON object with fields like "cache-control", "content-length", "content-type", "date", and "elapsed-time". The "Body" section shows a JSON array of hotel search results, with one item partially visible:

```
{ "@odata.context": "https://mydemo.search.windows.net/indexes('hotels-sample-index')/$metadata", "value": [ { "@search.score": 1, "HotelName": "Arcadia Resort & Restaurant", "Description": "The largest year-round resort in the area offering more of everything for your vacation needs.", "Address": { "City": "Seattle" } } ] }
```

3 - Visualize results

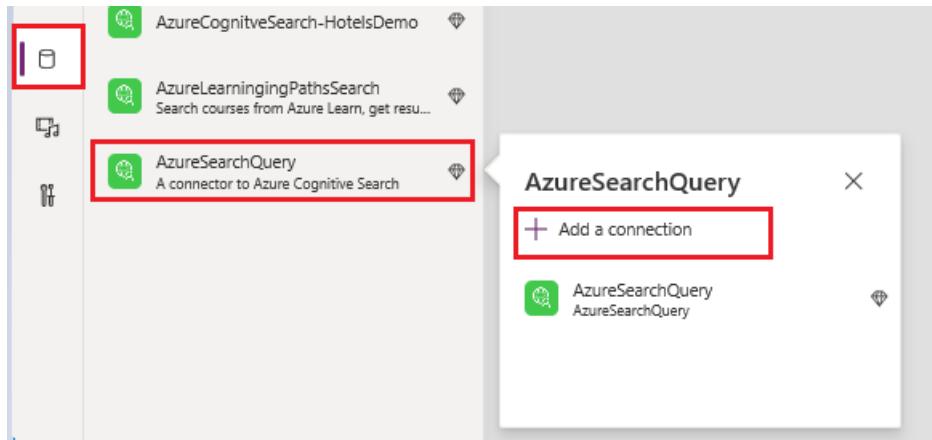
In this step, create a Power App with a search box, a search button, and a display area for the results. The Power App will connect to the recently created custom connector to get the data from Azure Search.

1. On the left, expand **Apps** > **+ New app** > **Canvas**.



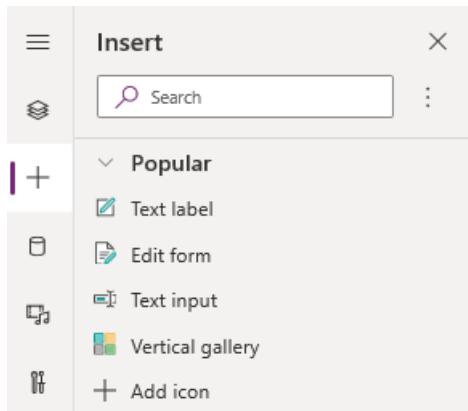
2. Select the type of application. For this tutorial, create a **Blank App** with the **Phone Layout**. The **Power Apps Studio** will appear.
3. Once in the studio, select the **Data Sources** tab, and click on the new Connector you have just created. In our case, it is called *AzureSearchQuery*. Click **Add a connection**.

Enter the query API key.



Now *AzureSearchQuery* is a data source that is available to be used from your application.

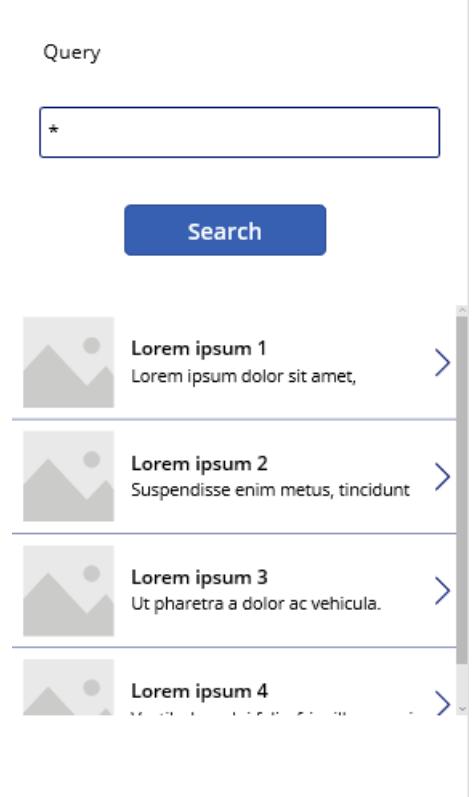
4. On the **Insert tab**, add a few controls to the canvas.



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*)

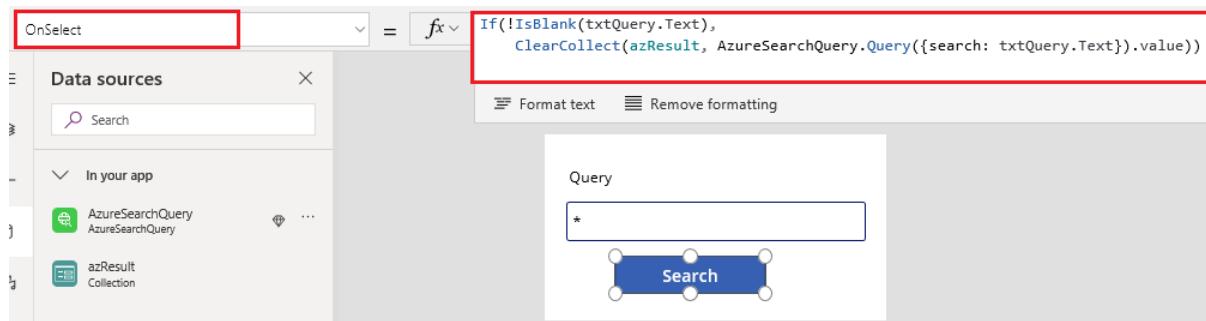
The canvas should look something like this:



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 will cause 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.

The screenshot shows the Power Apps builder interface. On the left, there's a preview of the app with a search bar and results for "Seattle". The results show two hotel entries: "Double Sanctuary Resort" and "Gacc Capital". On the right, the "Data" panel shows fields for "Body1" (Description, Address, Display value, Title2) and "Title2" (HotelName). The "Properties" panel shows the "Data source" set to "azResult" and the "Layout" set to "Title, subtitle, and body".

8. Press F5 to preview the app.

The screenshot shows the previewed app titled "Hotel Finder". It has a search bar with "+seattle" and a "Search" button. Below the search bar, it displays two hotel results: "Double Sanctuary Resort" and "Arcadia Resort & Restaurant". Each result includes the hotel name, location, and a brief description.

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

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.

Full text search in Azure Cognitive Search

10/4/2020 • 16 minutes to read • [Edit Online](#)

This article is for developers who need a deeper understanding of how 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 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.

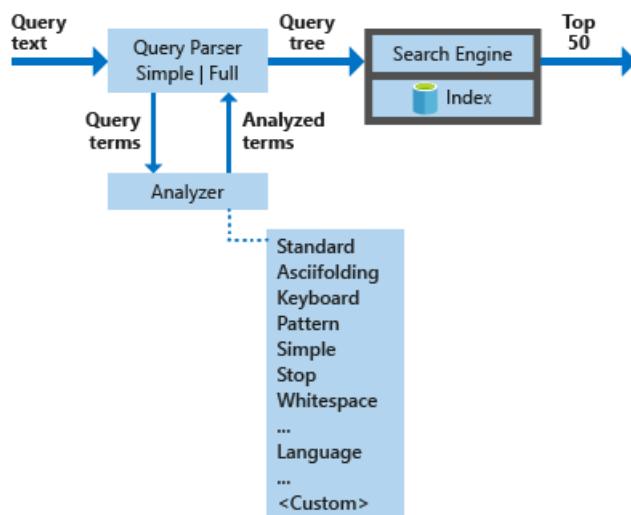
Architecture overview and diagram

Processing a full text search query starts with parsing the query text to extract search terms. The search engine uses an index to retrieve documents with matching terms. 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. 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.

Restated, query execution has four stages:

1. Query parsing
2. Lexical analysis
3. Document retrieval
4. Scoring

The diagram below illustrates the components used to process a search request.



KEY COMPONENTS

FUNCTIONAL DESCRIPTION

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 is 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:

1. Filters out 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 is not explicitly required (+).
3. Orders the resulting set of hotels by proximity to a given geography location, and then returned to the calling application.

The majority of 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\"",
```

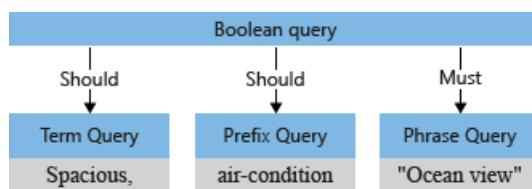
The query parser separates operators (such as `*` and `+` in the example) from search terms, and deconstructs the search query into *subqueries* of a supported type:

- *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?

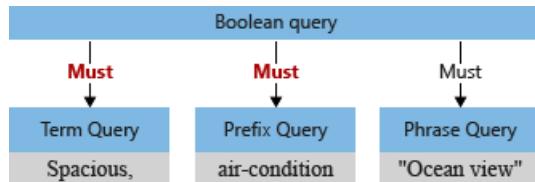
By default (`searchMode=any`), the search engine assumes the broader interpretation. Either field *should* be matched, reflecting "or" semantics. The initial query tree illustrated previously, with the two "should" operations, shows the default.

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* which 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

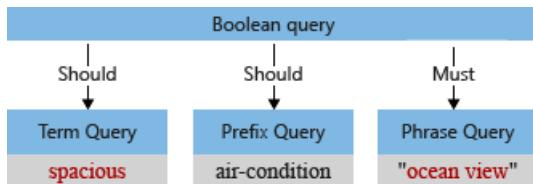
All of these operations tend to erase differences between the text input provided by the user and the terms stored in the index. Such operations go beyond text processing and require in-depth knowledge of the language itself. To add this layer of linguistic awareness, Azure Cognitive Search supports 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 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 is not 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:

```
{
  "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):

```
{
  "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:

```
{
  "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:

```
{  
    "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

TERM

DOCUMENT LIST

hotel	1, 3
ocean	4
playa	3
resort	3
retreat	4

In the title field, only *hote*/shows up in two documents: 1, 3.

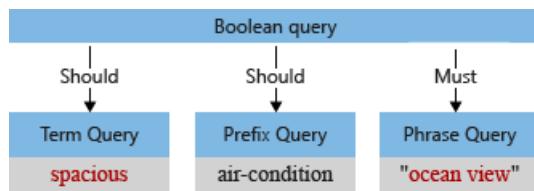
For the **description** field, the index is as follows:

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

TERM	DOCUMENT LIST
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 is split into two terms by the default analyzer. Recall that prefix queries, which contain partial terms, are not 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"
```

```
{
  "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 is 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.

Score 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 how to [add 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](#).

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 is 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 is not the best attribute for ordering documents if order stability is important. For example, given two documents with an identical score, there is 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 internet 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 a lot of 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 additional 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](#)

Indexers in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

An *indexer* in Azure Cognitive Search is a crawler that extracts searchable data and metadata from an external Azure data source and populates an index based on field-to-field mappings between the index and your data source. This approach is sometimes referred to as a 'pull model' because the service pulls data in without you having to write any code that adds data to an index.

Indexers are based on data source types or platforms, with individual indexers for SQL Server on Azure, Cosmos DB, Azure Table Storage and Blob Storage. Blob storage indexers have additional properties specific to blob content types.

You can use an indexer as the sole means for data ingestion, or use a combination of techniques that include the use of an indexer for loading just some of the fields in your index.

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.

Approaches for creating and managing indexers

You can create and manage indexers using these approaches:

- [Portal > Import Data Wizard](#)
- [Service REST API](#)
- [.NET SDK](#)

Initially, a new indexer is announced as a preview feature. Preview features are introduced in APIs (REST and .NET) and then integrated into the portal after graduating to general availability. If you're evaluating a new indexer, you should plan on writing code.

Permissions

All operations related to indexers, including GET requests for status or definitions, require an [admin api-key](#).

Supported data sources

Indexers crawl data stores on Azure.

- [Azure Blob Storage](#)
- [Azure Data Lake Storage Gen2](#) (in preview)
- [Azure Table Storage](#)
- [Azure Cosmos DB](#)
- [Azure SQL Database](#)
- [SQL Managed Instance](#)
- [SQL Server on Azure Virtual Machines](#)

Indexer Stages

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 Cosmos DB, change detection must be enabled.

For each of the document it ingests, an indexer implements or coordinates multiple steps, from document retrieval to a final search engine "handoff" for indexing. Optionally, an indexer is also instrumental in driving skillset execution and outputs, assuming a skillset is defined.



Stage 1: Document cracking

Document cracking is the process of opening files and extracting content. Depending on the type of data source, the indexer will try performing different operations to extract potentially indexable content.

Examples:

- When the document is a record in an [Azure SQL data source](#), the indexer will extract each of the fields for the record.
- When the document is a PDF file in an [Azure Blob Storage data source](#), the indexer will extract the text, images and metadata for the file.
- When the document is a record in a [Cosmos DB data source](#), the indexer will extract the fields and subfields from the 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 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. This step 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. Field mappings are optional.

Stage 3: Skillset execution

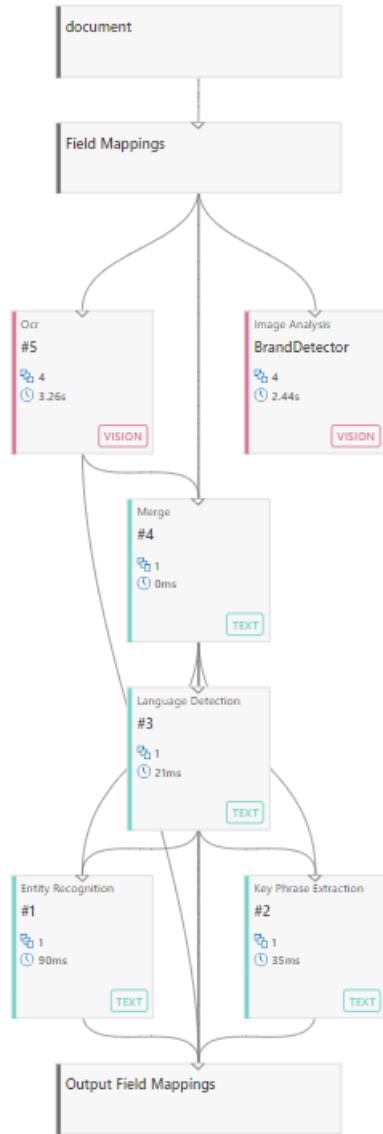
Skillset execution is an optional step that invokes built-in or custom AI processing. You might need it for optical character recognition (OCR) in the form of image analysis, or you might need language translation. Whatever the transformation, skillset execution is where enrichment occurs. If an indexer is a pipeline, you can think of a [skillset](#) as a "pipeline within the pipeline". A skillset has its own sequence of steps called skills.

Stage 4: Output field mappings

The output of a skillset is really a tree of information called the enriched document. Output field mappings allow you to select which parts of this tree to map into fields in your index. Learn how to [define output field mappings](#).

Just like field mappings that associate verbatim values from source to destination fields, output field mappings tell the indexer how to associate the transformed values in the enriched document to destination fields in the index. Unlike field mappings, which are considered optional, you will always need to define an output field mapping for any transformed content that needs to reside in an 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 configuration steps

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

An indexer obtains data source connection from a *data source* object. The data source definition provides a connection string and possibly credentials. Call the [Create Datasource REST API](#) or [DataSource class](#) to create the resource.

Data sources are configured and managed independently of the indexers that use them, which means a data source can be used by multiple indexers to load more than one index at a time.

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 with fields that match those in your external data source. Fields need to match by name and data type. For more information about structuring an index, see [Create an Index \(Azure Cognitive Search REST API\)](#) or [Index class](#). For help with field associations, see [Field mappings in Azure Cognitive Search indexers](#).

TIP

Although indexers cannot generate an index for you, the **Import data** wizard in the portal can help. In most cases, the wizard can infer an index schema from existing metadata in the source, presenting a preliminary index schema which you can edit in-line while the wizard is active. Once the index is created on the service, further edits in the portal are mostly limited to adding new fields. Consider the wizard for creating, but not revising, an index. For hands-on learning, step through the [portal walkthrough](#).

Step 3: Create and schedule the indexer

The indexer definition is a construct that brings together all of the elements related to data ingestion. Required elements include a data source and index. Optional elements include a schedule and field mappings. Field mapping are only optional if source fields and index fields clearly correspond. For more information about structuring an indexer, see [Create Indexer \(Azure Cognitive Search REST API\)](#).

Run indexers on-demand

While it's common to schedule indexing, an indexer can also be invoked on demand using the [Run command](#):

```
POST https://[service name].search.windows.net/indexers/[indexer name]/run?api-version=2020-06-30  
api-key: [Search service admin key]
```

NOTE

When Run API returns successfully, the indexer invocation has been scheduled, but the actual processing happens asynchronously.

You can monitor the indexer status in the portal or through Get Indexer Status API.

Get indexer status

You can retrieve the status and execution history of an indexer through the [Get Indexer Status command](#):

```
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.

```
{  
    "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 completed executions, which are sorted in reverse chronological order (so the latest execution comes first in the response).

Next steps

Now that you have the basic idea, the next step is to review requirements and tasks specific to each data source type.

- [Azure SQL Database, SQL Managed Instance, or SQL Server on an Azure virtual machine](#)
- [Azure Cosmos DB](#)
- [Azure Blob Storage](#)
- [Azure Table Storage](#)
- [Indexing CSV blobs using the Azure Cognitive Search Blob indexer](#)
- [Indexing JSON blobs with Azure Cognitive Search Blob indexer](#)

AI enrichment in Azure Cognitive Search

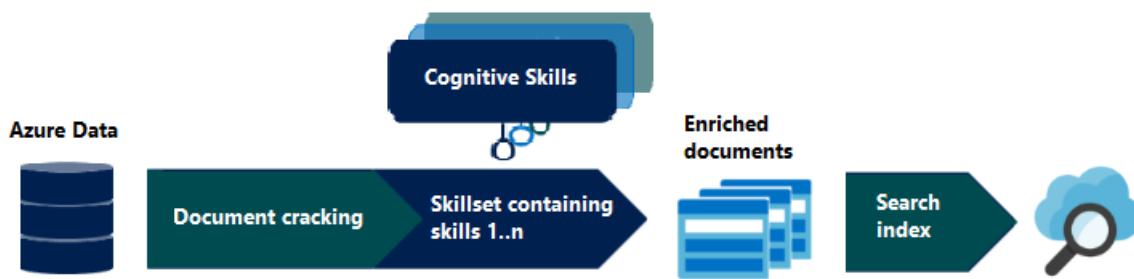
10/4/2020 • 7 minutes to read • [Edit Online](#)

AI enrichment is an extension of [indexers](#) that can be used to extract text from images, blobs, and other unstructured data sources. Enrichment and extraction make your content more searchable in indexer output objects, either a [search index](#) or a [knowledge store](#).

Extraction and enrichment are implemented using *cognitive skills* attached to the indexer-driven pipeline. You can use built-in skills from Microsoft or embed external processing into a [custom skill](#) that you create. Examples of a custom skill might be a custom entity module or document classifier targeting a specific domain such as finance, scientific publications, or medicine.

Built-in skills fall into these categories:

- **Natural language processing** skills include [entity recognition](#), [language detection](#), [key phrase extraction](#), text manipulation, [sentiment detection](#), and [PII detection](#). With these skills, unstructured text is mapped as searchable and filterable fields in an index.
- **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, making it searchable using the query capabilities of Azure Cognitive Search.



Built-in skills in Azure Cognitive Search are based on pre-trained machine learning models in Cognitive Services APIs: [Computer Vision](#) and [Text Analytics](#). You can attach a Cognitive Services resource if you want to leverage these resources during content processing.

Natural language and image processing is applied during the data ingestion phase, with results becoming part of a document's composition in a searchable index in Azure Cognitive Search. Data is sourced as an Azure data set and then pushed through an indexing pipeline using whichever [built-in skills](#) you need.

When to use AI enrichment

You should consider using built-in cognitive skills if your raw content is unstructured text, image content, or content that needs language detection and translation. Applying AI through the built-in cognitive skills can unlock this content, increasing its value and utility in your search and data science apps.

Additionally, you might consider adding a custom skill if you have open-source, third-party, or first-party code that you'd like to integrate into the pipeline. Classification models that identify salient characteristics of various document types fall into this category, but any package that adds value to your content could be used.

More about built-in skills

A [skillset](#) that's assembled using built-in skills is well suited for the following application scenarios:

- Scanned documents (JPEG) that you want to make full-text searchable. You can attach an optical character recognition (OCR) skill to identify, extract, and ingest text from JPEG files.
- PDFs with combined image and text. Text in PDFs can be extracted during indexing without the use of enrichment steps, but the addition of image and natural language processing can often produce a better outcome than a standard indexing provides.
- Multi-lingual content against which you want to apply language detection and possibly text translation.
- Unstructured or semi-structured documents containing content that has inherent meaning or context that is hidden in the larger document.

Blobs in particular often contain a large body of content that is packed into a single "field". By attaching image and natural language processing skills to an indexer, you can create new information that is extant in the raw content, but not otherwise surfaced as distinct fields. Some ready-to-use built-in cognitive skills that can help: key phrase extraction, sentiment analysis, and entity recognition (people, organizations, and locations).

Additionally, built-in skills can also be used to restructure content through text split, merge, and shape operations.

More about custom skills

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](#), integration of the [Bing Entity Search API](#), and [custom entity recognition](#).

Steps in an enrichment pipeline

An enrichment pipeline is based on [indexers](#). Indexers populate an index based on field-to-field mappings between the index and your data source for document cracking. Skills, now attached to indexers, intercept and enrich documents according to the skillset(s) you define. Once indexed, you can access content via search requests through all [query types supported by Azure Cognitive Search](#). If you are new to indexers, this section walks you through the steps.

Step 1: Connection and document cracking phase

At the start of the pipeline, you have unstructured text or non-text content (such as images, scanned documents, or JPEG files). Data must exist in an Azure data storage service that can be accessed by an indexer. Indexers can "crack" source documents to extract text from source data. Document cracking is the process of extracting or creating text content from non-text sources during indexing.

Azure Data



Supported sources include Azure blob storage, Azure table storage, Azure SQL Database, and Azure Cosmos DB. Text-based content can be extracted from the following file types: PDFs, Word, PowerPoint, CSV files. For the full list, see [Supported formats](#). Indexing takes time so start with a small, representative data set and then build it up incrementally as your solution matures.

Step 2: Cognitive skills and enrichment phase

Enrichment is performed with *cognitive skills* performing atomic operations. For example, once you have cracked a PDF, you can apply entity recognition, language detection, or key phrase extraction to produce new fields in your index that are not available natively in the source. Altogether, the collection of skills used in your pipeline is called a *skillset*.



A skillset is based on [built-in cognitive skills](#) or [custom skills](#) you provide and connect to the skillset. A skillset can be minimal or highly complex, and determines not only the type of processing, but also the order of operations. A skillset plus the field mappings defined as part of an indexer fully specifies the enrichment pipeline. For more information about pulling all of these pieces together, see [Define a skillset](#).

Internally, the pipeline generates a collection of enriched documents. You can decide which parts of the enriched documents should be mapped to indexable fields in your search index. For example, if you applied the key phrase extraction and the entity recognition skills, those new fields would become part of the enriched document, and can be mapped to fields on your index. See [Annotations](#) to learn more about input/output formations.

Add a knowledgeStore element to save enrichments

[Search REST api-version=2020-06-30](#) extends skillsets with a `knowledgeStore` definition that provides an Azure storage connection and projections that describe how the enrichments are stored. This is in addition to your index. In a standard AI pipeline, enriched documents are transitory, used only during indexing and then discarded. With knowledge store, enriched documents are preserved. For more information, see [Knowledge store](#).

Step 3: Search index and query-based access

When processing is finished, you have a search index consisting of enriched documents, fully text-searchable in Azure Cognitive Search. [Querying the index](#) is how developers and users access the enriched content generated by the pipeline.



The index is like any other you might create for Azure Cognitive Search: you can supplement with custom analyzers, invoke fuzzy search queries, add filtered search, or experiment with scoring profiles to reshape the search results.

Indexes are generated from an index schema that defines the fields, attributes, and other constructs attached to a specific index, such as scoring profiles and synonym maps. Once an index is defined and populated, you can index incrementally to pick up new and updated source documents. Certain modifications require a full rebuild. You should use a small data set until the schema design is stable. For more information, see [How to rebuild an index](#).

Checklist: A typical workflow

1. Subset your Azure source data into a representative sample. Indexing takes time so start with a small, representative data set and then build it up incrementally as your solution matures.
2. Create a [data source object](#) in Azure Cognitive Search to provide a connection string for data retrieval.
3. Create a [skillset](#) with enrichment steps.
4. Define the [index schema](#). The *Fields* collection includes fields from source data. You should also stub out additional fields to hold generated values for content created during enrichment.
5. Define the [indexer](#) referencing the data source, skillset, and index.
6. Within the indexer, add *outputFieldMappings*. This section maps output from the skillset (in step 3) to

the inputs fields in the index schema (in step 4).

7. Send *Create Indexer* request you just created (a POST request with an indexer definition in the request body) to express the indexer in Azure Cognitive Search. This step is how you run the indexer, invoking the pipeline.
8. Run queries to evaluate results and modify code to update skillsets, schema, or indexer configuration.
9. [Reset the indexer](#) before rebuilding the pipeline.

Next steps

- [AI enrichment documentation links](#)
- [Example: Creating a custom skill for AI enrichment \(C#\)](#)
- [Quickstart: Try AI enrichment in a portal walk-through](#)
- [Tutorial: Learn about the AI enrichment APIs](#)
- [Knowledge store](#)
- [Create a knowledge store in REST](#)
- [Troubleshooting tips](#)

Skillset concepts in Azure Cognitive Search

10/4/2020 • 12 minutes to read • [Edit Online](#)

This article is for developers who need a deeper understanding of skillset concepts and composition, and assumes familiarity with the AI enrichment process. If you are new to this concept, start with [AI enrichment in Azure Cognitive Search](#).

Introducing skillsets

A skillset is a reusable resource in Azure Cognitive Search that is attached to an indexer, and it specifies a collection of skills used for analyzing, transforming, and enriching text or image content during indexing. Skills have inputs and outputs, and often the output of one skill becomes the input of another in a chain or sequence of processes.

A skillset has three main properties:

- `skills`, an unordered collection of skills for which the platform determines the sequence of execution based on the inputs required for each skill.
- `cognitiveServices`, the key of a Cognitive Services resource that performs image and text processing for skillsets that include built-in skills.
- `knowledgeStore`, (optional) an Azure Storage account where your enriched documents will be projected.
Enriched documents are also consumed by search indexes.

Skillsets are authored in JSON. The following example is a slightly simplified version of this [hotel-reviews skillset](#), used to illustrate concepts in this article.

The first two skills are shown below:

- Skill #1 is a [Text Split skill](#) that accepts the contents of the "reviews_text" field as input, and splits that content into "pages" of 5000 characters as output.
- 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.

```
{
  "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": "score",
          "targetName": "Sentiment"
        }
      ]
    },
    "cognitiveServices": null,
    "knowledgeStore": { }
  }
}
```

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 syntax.

Enrichment tree

In the progression of [steps in an enrichment pipeline](#), content processing follows the *document cracking* phase where text and images are extracted from the source. Image content can then be routed to skills that specify image processing, while text content is queued for text processing. For source documents that contain large quantities of text, you can set a *parsing mode* on the indexer to segment text into smaller chunks for more optimal processing.

Once a document is in the enrichment pipeline, it is represented as a tree of content and associated enrichments.

This tree is instantiated as the output of document cracking. The enrichment tree format enables the enrichment pipeline to attach metadata to even primitive data types, it is not a valid JSON object but can be projected into a valid JSON format. The following table shows the state of a document entering into the enrichment pipeline:

DATA SOURCE\PARSING MODE	DEFAULT	JSON, JSON LINES & CSV
Blob Storage	/document/content /document/normalized_images/* ...	/document/{key1} /document/{key2} ...
SQL	/document/{column1} /document/{column2} ...	N/A
Cosmos DB	/document/{key1} /document/{key2} ...	N/A

As skills execute, they add new nodes to the enrichment tree. These new nodes may then be used as inputs for downstream skills, projecting to the knowledge store, or mapping to index fields. Enrichments aren't mutable: once created, nodes cannot 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 only a subset of the enrichments to the index or the knowledge store.

Context

Each skill requires a context. A context determines:

- The number of times the skill executes, based on the nodes selected. For context values of type collection, adding an `/*` at the end will result in the skill being invoked once for each instance in the collection.
- 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.

CONTEXT	INPUT	SHAPE OF INPUT	SKILL INVOCATION
<code>/document/countries/*</code>	<code>/document/countries/*/states/*</code>	A list of all ZIP codes in the country/region	Once per country/region
<code>/document/countries/*/states/*</code>	<code>/document/countries//states/*/zipcodes/*</code>	A list of ZIP codes in the state	Once per combination of country/region and state

Generate enriched data

Using the [hotel reviews skillset](#) as a reference point, we are going to look at:

- How the enrichment tree evolves with the execution of each skill
- How the context and inputs work to determine how many times a skill executes
- What the shape of the input is based on the context

A "document" within the enrichment process represents a single row (a hotel review) within the `hotel_reviews.csv` source file.

Skill #1: Split skill

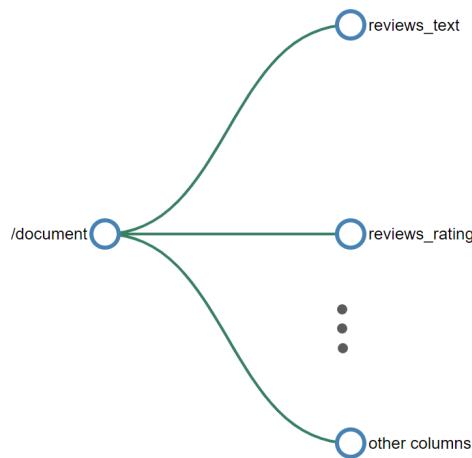
When source content consists of large chunks of text, it's helpful to break it into smaller components for greater accuracy of language, sentiment, and key phrase detection. There are two grains available: pages and sentences. A page consists of approximately 5000 characters.

A text split skill is typically first in a skillset.

```
"@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"
}]
```

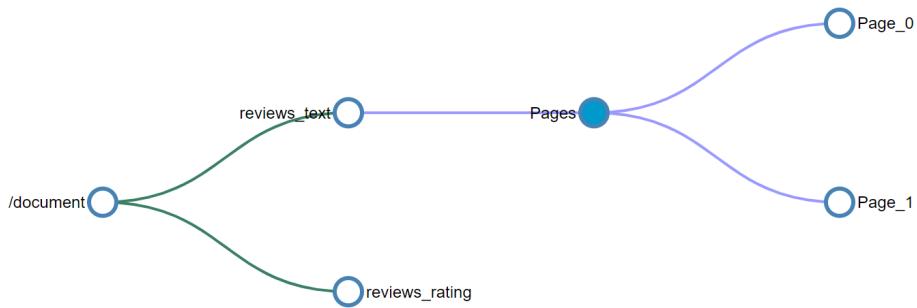
With the skill context of `"/document/reviews_text"`, the split skill will execute once for the `reviews_text`. The skill output is a list where the `reviews_text` is chunked into 5000 character segments. The output from the split skill is named `pages` and it is added to the enrichment tree. The `targetName` 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. Conceptually, the tree looks as follows:



The root node for all enrichments is `"/document"`. When working with blob indexers, the `"/document"` node will have child nodes of `"/document/content"` and `"/document/normalized_images"`. When working with CSV data, as we are in this example, the column names will map to nodes beneath `"/document"`.

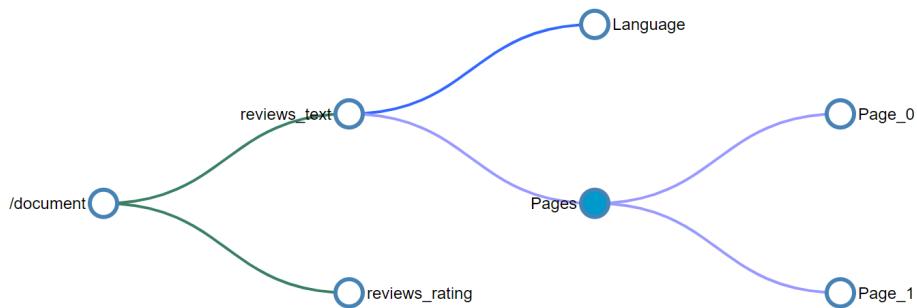
To access any of the enrichments added to a node by a skill, the full path for the enrichment is needed. For example, if you want to use the text from the `pages` node as an input to another skill, you will need to specify it as `"/document/reviews_text/pages/*"`.



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 is then passed to key phrase extraction and sentiment detection, taking language into consideration when detecting sentiment and phrases.

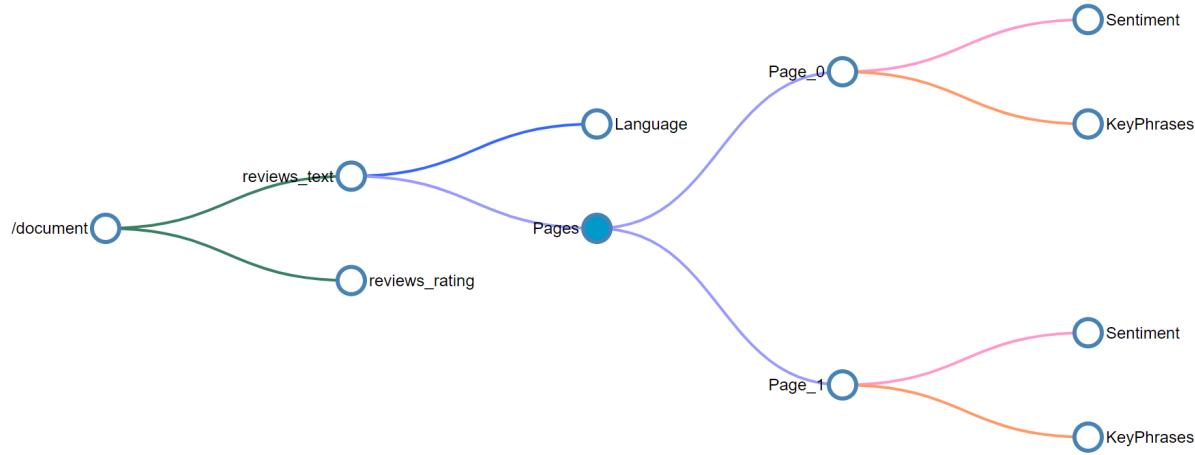
While the language detection skill is the third (skill #3) skill defined in the skillset, it is the next skill to execute. Since it is not blocked by requiring any inputs, it will execute in parallel with the previous skill. Like the split skill that preceded it, the language detection skill is also invoked once for each document. The enrichment tree now has a new node for language.



Skill #3: Key phrases skill

Given the context of `/document/reviews_text/pages/*` the key phrases skill 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 will not be part of the object returned when selecting the parent node.

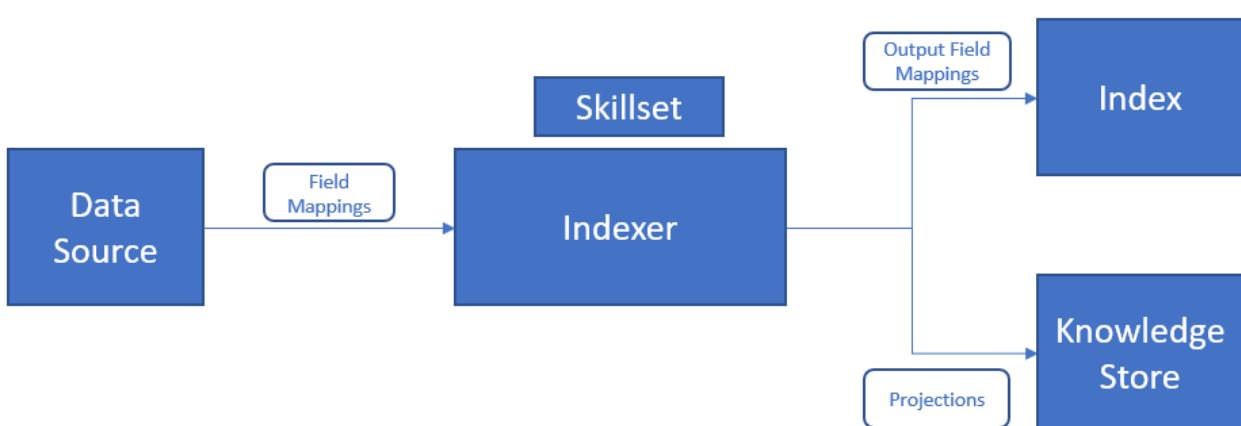
Save enrichments

In Azure Cognitive Search, an indexer saves the output it creates. One of the outputs is always a [searchable index](#). Specifying an index is a requirement, and when you attach a skillset, the data ingested by an index includes the substance of the enrichments. Usually, the outputs of specific skills, such as key phrases or sentiment scores, are ingested into the index in a field created for that purpose.

Optionally, an indexer can also send the output to a [knowledge store](#) for consumption in other tools or processes. A knowledge store is defined as part of the skillset. Its definition determines whether your enriched documents are projected as tables or objects (files or blobs). Tabular projections are well-suited for interactive analysis in tools like Power BI, whereas files and blobs are typically used in data science or similar processes. In this section, you'll learn how skillset composition can shape the tables or objects you want to project.

Projections

For content that targets a knowledge store, you will want to consider how the content is structured. *Projection* is the process of selecting the nodes from the enrichment tree and creating a physical expression of them in the knowledge store. Projections are custom shapes of the document (content and enrichments) that can be output as either table or object projections. To learn more about working with projections, see [working with projections](#).



SourceContext

The `sourceContext` element is only used in skill inputs and projections. It is used to construct multi-level, nested

objects. You may need to create a new object to either pass it as an input to a skill or project into the knowledge store. As enrichment nodes may not be a valid JSON object in the enrichment tree and referencing a node in the tree only returns that state of the node when it was created, using the enrichments as skill inputs or projections requires you to create a well formed JSON object. The `sourceContext` enables you to construct a hierarchical, anonymous type object, which would require multiple skills if you were only using the context.

Using `sourceContext` is shown in the following examples. Look at the skill output that generated an enrichment to determine if it is a valid JSON object and not a primitive type.

Slicing projections

When defining a table projection group, a single node in the enrichment tree can be sliced into multiple related tables. If you add a table with a source path that is a child of an existing table projection, the resulting child node will not be a child of the existing table projection, but instead will be projected into the new, related, table. This slicing technique allows you to define a single node in a shaper skill that can be the source for all your table projections.

Shaping projections

There are two ways to define a projection:

- Use the Text Shaper skill to create a new node that is the root node for all the enrichments you are projecting. Then, in your projections, you would only reference the output of the shaper skill.
- Use an inline shape a projection within the projection definition itself.

The shaper approach is more verbose than inline shaping but ensures that all the mutations of the enrichment tree are contained within the skills 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 is defined. The approaches can be used together or separately. The skillset created for you in the portal workflow contains both. It uses a shaper skill for the table projections, but also uses inline shaping to project the key phrases table.

To extend the example, you could choose to remove the inline shaping and use a shaper skill to create a new node for the key phrases. To create a shape projected into three tables, namely, `hotelReviewsDocument`, `hotelReviewsPages`, and `hotelReviewsKeyPhrases`, the two options are described in the following sections.

Shaper skill and projection

This

NOTE

Some of the columns from the document table have been removed from this example for brevity.

```
{  
    "@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": []  
        }  
    ]  
}
```

```

        "inputs": []
    },
    {
        "name": "AzureSearch_DocumentKey",
        "source": "/document/AzureSearch_DocumentKey",
        "sourceContext": null,
        "inputs": []
    },
    {
        "name": "pages",
        "source": null,
        "sourceContext": "/document/reviews_text/pages/*",
        "inputs": [
            {
                "name": "SentimentScore",
                "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"
        }
    ]
}

```

With the `tableprojection` node defined in the `outputs` section above, we can now use the slicing feature to project parts of the `tableprojection` node into different tables:

NOTE

This is only a snippet of the projection within the knowledge store configuration.

```
"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 shaping projections

The inline shaping approach does not require a shaper skill as all shapes needed for the projections are created at the time they are needed. To project the same data as the previous example, the inline projection option would look like this:

```

"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": "SentimentScore",
                        "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.

Next steps

As a next step, create your first skillset with cognitive skills.

[Create your first skillset.](#)

Debug sessions in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

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 push changes to a production skillset in the AI enrichment pipeline.

IMPORTANT

Debug sessions is a preview feature provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Using debug sessions

When you start a session, the service creates a copy of the skillset, indexer, and index where a single document is used to test the skillset. Changes made within the session are saved to the session. The changes made within the debug session will not affect the production skillset unless the changes are committed to it. Committing changes will overwrite the production skillset.

During a debug session you can edit a skillset, inspect, and validate each node in the enrichment tree in the context of a specific document. Once the enrichment pipeline issues are resolved, changes can be saved to the session and committed to the skillset in production.

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.

Creating a debug session

To start a debug session you must have an existing AI enrichment pipeline including; a data source, a skillset, an indexer, and an index. To configure a debug session, you need to name the session, and provide a general-purpose storage account that will be used to cache the skill executions during the indexer run. You will also need to select the indexer that will be running. The indexer has references stored to the data source, skillset, and index. The debug session will default to the first document in the data source or you can specify a document in the data source to step through.

new-debug-session
Debug session

Save Session Refresh Delete | Run Commit changes... download

Definition AI Enrichments Errors/Warnings

Debug session name * ✓

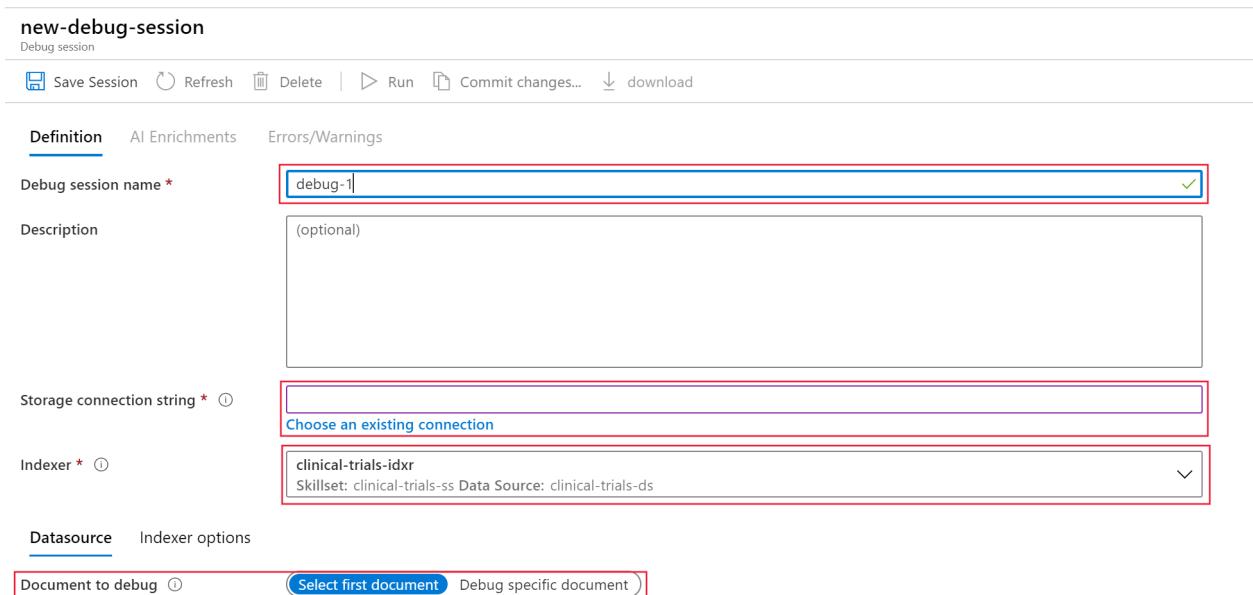
Description

Storage connection string *

Indexer *

Datasource Indexer options

Document to debug Select first document Debug specific document



Debug session features

The debug session begins by executing the skillset on the selected document. The debug session will record additional metadata associated with each operation in the skillset. The metadata created by the pipeline's skill executions, informs the following set of features that are then used to help identify and fix issues with the skillset.

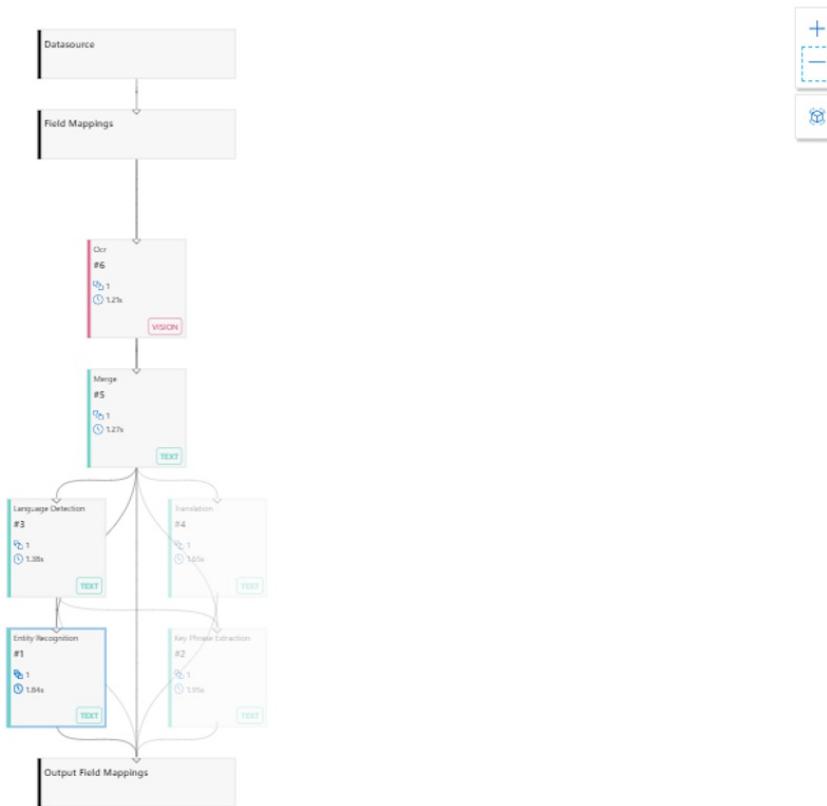
AI Enrichments

As skills execute a tree of enrichments, representing the document, grows. Using a tree to visualize the outputs of skills or enrichments provides a comprehensive look at all the enrichments performed. You can look across the entire document and inspect each node of the enrichment tree. This perspective makes it easier to shape objects. This format also provides visual cues to the type, path, and contents of each node in the tree.

Skill Graph

The **Skill Graph** view provides a hierarchical, visual representation of the skillset. The graph is a top to bottom representation of the order in which the skills are executed. Skills that are dependent upon the output of other skills will be shown lower in the graph. Skills at the same level in the hierarchy can execute in parallel.

Selecting a skill in the graph will highlight the skills connected to it, the nodes that create its inputs and the nodes that accept its outputs. Each skill node displays its type, errors or warnings, and execution counts. The **Skill Graph** is where you will select which skill to debug or enhance. When you select a skill its details will be displayed in the skill details pane to the right of the graph.



Skill details

The skill details pane displays a set of areas for working with a specific skill, when that skill is highlighted in the **Skill Graph**. You can review and edit the details of the skill's settings. The skill's JSON definition is provided. The details of the skill's execution and the errors and warnings are also displayed. The **Skill Settings** tab & **Skill JSON Editor** allow for direct edits to the skill. The `</>` opens a window for viewing and editing the expressions of the skills inputs and outputs.

Nested input controls in the skill settings window 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.

Skill execution history

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 each document. In the skill details pane the **Executions** tab 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. This allows identification of the root cause of a problem that may manifest in a downstream skill.

When a potential issue is identified, the execution history displays links to the skills that generated the specific inputs, providing a stack-trace like feature. Clicking on the skill associated with an input, navigates to the skill to fix any bugs or continue to trace the specific issue.

When building a custom skill or debugging an error with a custom skill, there is the option to generate a request for a skill invocation in the execution history.

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.

The screenshot shows the 'AI Enrichments' tab selected in the top navigation bar. A 'Skill Graph' pane is visible on the left, showing a path from '/document/path' to 'Enriched Data Structure'. The main pane displays a table of enriched paths:

Path	Output	Originating Source
document	{"normalized_images": [{"\$type": "file", "url": "@trim..."}]}	#1
merged_content	"\n\n[n[Study of BMN 110 in ...]	#5
keyphrases	["Study of BMN", "Syndrome", "Pediatric Patients", "Y...	#2
locations	["IVA"]	#1
translated_text	"\n[image: image0.jpg] Study of BMN 110 in Pedia...	#4
entities	[{"name": "BMN", "wikipediaId": null, "wikipediaLang...	#1
organizations	["BMN"]	#1
language	"en"	#3
normalized_images		
#		
layoutText	[{"language": "en", "text": ["@trimmed": true, "display": ...]}]	#6
text	[{@trimmed": true, "display": "Study of BMN 110 in ...}]]	#6

To the right, the 'Expression evaluator' panel is open, showing the context path '/document' and the expression '/document/merged_content/keyphrases'. The resulting value is a list of keyphrases:

Value
1 "Study of BMN", 2 "Syndrome", 3 "Pediatric Patients", 4 "Years of Age", 5 "Morquio", 6 "dose of study drug", 7 "MPS IVA", 8 "Mucopolysaccharidosis IVA", 9 "wk infusions of BMN", 10 "open-label Phase", 11 "safety", 12 "time of administration", 13 "efficacy of weekly", 14 "

Buttons for 'Save', 'Discard', 'Delete', and a '+' icon are at the top right of the pane.

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.

Errors/Warnings

This window displays all of the errors and warnings the skillset produces as it is executed against the document in the debug session.

Limitations

Debug sessions work with all generally available data sources and most preview data sources. The MongoDB API (preview) and Cassandra API (preview) of Cosmos DB are currently not supported.

Next steps

Now that you understand the elements of Debug sessions try the tutorial for a hands-on experience.

[Explore Debug sessions feature tutorial](#)

Knowledge store in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

Knowledge store is a feature of Azure Cognitive Search that persists output from an [AI enrichment pipeline](#) for independent analysis or downstream processing. An *enriched document* is a pipeline's output, created from content that has been extracted, structured, and analyzed using AI processes. In a standard AI pipeline, enriched documents are transitory, used only during indexing and then discarded. Choosing to create a knowledge store will allow you to preserve the enriched documents.

If you have used cognitive skills in the past, you already know that *skillsets* move a document through a sequence of enrichments. The outcome can be a search index, or projections in a knowledge store. The two outputs, search index and knowledge store, are products of the same pipeline; derived from the same inputs, but resulting in output that is structured, stored, and used in very different ways.

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.

Benefits of knowledge store

A knowledge store gives you structure, context, and actual content - gleaned from unstructured and semi-structured data files like blobs, image files that have undergone analysis, or even structured data, reshaped into new forms. In a [step-by-step walkthrough](#), you can see first-hand how a dense JSON document is partitioned out into substructures, reconstituted into new structures, and otherwise made available for downstream processes like machine learning and data science workloads.

Although it's useful to see what an AI enrichment pipeline can produce, the real potential of a knowledge store is the ability to reshape data. You might start with a basic skillset, and then iterate over it to add increasing levels of structure, which you can then combine into new structures, consumable in other apps besides Azure Cognitive Search.

Enumerated, the benefits of knowledge store include the following:

- Consume enriched documents in [analytics and reporting tools](#) other than search. Power BI with Power Query is a compelling choice, but any tool or app that can connect to Azure Storage can pull from a knowledge store that you create.
- Refine an AI-indexing pipeline while debugging steps and skillset definitions. A knowledge store shows you the product of a skillset definition in an AI-indexing pipeline. You can use those results to design a better skillset because you can see exactly what the enrichments look like. You can use [Storage Explorer](#) in Azure Storage to view the contents of a knowledge store.
- Shape the data into new forms. The reshaping is codified in skillsets, but the point is that a skillset can now provide this capability. The [Shaper skill](#) in Azure Cognitive Search has been extended to accommodate this task. Reshaping allows you to define a projection that aligns with your intended use of the data while preserving relationships.

NOTE

New to AI enrichment and cognitive skills? Azure Cognitive Search integrates with Cognitive Services Vision and Language features to extract and enrich source data using Optical Character Recognition (OCR) over image files, entity recognition and key phrase extraction from text files, and more. For more information, see [AI enrichment in Azure Cognitive Search](#).

Physical storage

The physical expression of a knowledge store is articulated through the `projections` element of a `knowledgeStore` definition in a Skillset. The projection defines a structure of the output so that it matches your intended use.

Projections can be articulated as tables, objects, or files.

```
"knowledgeStore": {  
    "storageConnectionString": "<YOUR-AZURE-STORAGE-ACCOUNT-CONNECTION-STRING>",  
    "projections": [  
        {  
            "tables": [ ],  
            "objects": [ ],  
            "files": [ ]  
        },  
        {  
            "tables": [ ],  
            "objects": [ ],  
            "files": [ ]  
        }  
    ]  
}
```

The type of projection you specify in this structure determines the type of storage used by knowledge store.

- Table storage is used when you define `tables`. 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` 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.
- Blob storage is used when you define `objects` or `files`. The physical representation of an `object` is a hierarchical JSON structure that represents an enriched document. A `file` is an image extracted from a document, transferred intact to Blob storage.

A single projection object contains one set of `tables`, `objects`, `files`, and for many scenarios, creating one projection might be enough.

However, it is possible to create multiple sets of `table - object - file` projections, and you might do that if you want different data relationships. Within a set, data is related, assuming those relationships exist and can be detected. If you create additional sets, the documents in each group are never related. An example of using multiple projection groups might be if you want the same data projected for use with your online system and it needs to be represented a specific way, you also want the same data projected for use in a data science pipeline that is represented differently.

Requirements

[Azure Storage](#) is required. It provides physical storage. You can use Blob storage, Table storage or both. Blob storage is used for intact enriched documents, usually when the output is going to downstream processes. Table storage is for slices of enriched documents, commonly used for analysis and reporting.

[Skillset](#) is required. It contains the `knowledgeStore` definition, and it determines the structure and composition of an enriched document. You cannot create a knowledge store using an empty skillset. You must have at least one skill in a skillset.

[Indexer](#) is required. A skillset is invoked by an indexer, which drives the execution. Indexers come with their own set of requirements and attributes. Several of these attributes have a direct bearing on a knowledge store:

- Indexers require a [supported Azure data source](#) (the pipeline that ultimately creates the knowledge store starts by pulling data from a supported source on Azure).
- Indexers require a search index. An indexer requires that you provide an index schema, even if you never plan to use it. A minimal index has one string field, designated as the key.
- Indexers provide optional field mappings, used to alias a source field to a destination field. If a default field mapping needs modification (to use a different name or type), you can create a [field mapping](#) within an indexer. For knowledge store output, the destination can be a field in a blob object or table.
- Indexers have schedules and other properties, such as change detection mechanisms provided by various data sources, can also be applied to a knowledge store. For example, you can [schedule](#) enrichment at regular intervals to refresh the contents.

How to create a knowledge store

To create knowledge store, use the portal or the REST API (`api-version=2020-06-30`).

Use the Azure portal

The [Import data](#) wizard includes options for creating a knowledge store. For initial exploration, [create your first knowledge store in four steps](#).

1. Select a supported data source.
2. Specify enrichment: attach a resource, select skills, and specify a knowledge store.
3. Create an index schema. The wizard requires it and can infer one for you.
4. Run the wizard. Extraction, enrichment, and storage occur in this last step.

Use Create Skillset (REST API)]

A `knowledgeStore` is defined within a [skillset](#), which in turn is invoked by an [indexer](#). During enrichment, Azure Cognitive Search creates a space in your Azure Storage account and projects the enriched documents as blobs or into tables, depending on your configuration.

The REST API is one mechanism by which you can create a knowledge store programmatically. An easy way to explore is [create your first knowledge store using Postman and the REST API](#).

How to connect with tools and apps

Once the enrichments exist in storage, any tool or technology that connects to Azure Blob or Table storage can be used to explore, analyze, or consume the contents. The following list is a start:

- [Storage Explorer](#) 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.

API reference

REST API version `2020-06-30` provides knowledge store through additional definitions on skillsets. In addition to the reference, see [Create a knowledge store using Postman](#) for details on how to call the APIs.

- [Create Skillset \(api-version=2020-06-30\)](#)
- [Update Skillset \(api-version=2020-06-30\)](#)

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.

[Create a knowledge store using Postman and REST](#)

To learn more about projections, the capabilities and how you [define them in a skillset](#)

[Projections in a knowledge store](#)

For a tutorial covering advanced projections concepts like slicing, inline shaping and relationships, start with [define projections in a knowledge store](#)

[Define projections in a knowledge store](#)

Knowledge store "projections" in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

Azure Cognitive Search enables content enrichment through built-in cognitive skills and custom skills as part of indexing. Enrichments create new information where none previously existed: extracting information from images, detecting sentiment, key phrases, and entities from text, to name a few. Enrichments also add structure to undifferentiated text. All of these processes result in documents that make full text search more effective. In many instances, enriched documents are useful for scenarios other than search, such as for knowledge mining.

Projections, a component of [knowledge store](#), are views of enriched documents that can be saved to physical storage for knowledge mining purposes. A projection lets you "project" your data into a shape that aligns with your needs, preserving relationships so that tools like Power BI can read the data with no additional effort.

Projections can be tabular, with data stored in rows and columns in Azure Table storage, or JSON objects stored in Azure Blob storage. You can define multiple projections of your data as it is being enriched. Multiple projections are useful when you want the same data shaped differently for individual use cases.

The knowledge store supports three types of projections:

- **Tables:** For data that's best represented as rows and columns, table projections allow you to define a schematized shape or projection in Table storage. Only valid JSON objects can be projected as tables, the enriched document can contain nodes that are not named JSON objects and when projecting these objects, create a valid JSON object with a shaper skill or inline shaping.
- **Objects:** When you need a JSON representation of your data and enrichments, object projections are saved as blobs. Only valid JSON objects can be projected as objects, the enriched document can contain nodes that are not named JSON objects and when projecting these objects, create a valid JSON object with a shaper skill or inline shaping.
- **Files:** When you need to save the images extracted from the documents, file projections allow you to save the normalized images to blob storage.

To see projections defined in context, step through [Create a knowledge store in REST](#).

Projection groups

In some cases, you will need to project your enriched data in different shapes to meet different objectives. The knowledge store allows you to define multiple groups of projections. Projection groups have the following key characteristics of mutual exclusivity and relatedness.

Mutual exclusivity

All content projected into a single group is independent of data projected into other projection groups. This independence implies that you can have the same data shaped differently, yet repeated in each projection group.

Relatedness

Projection groups now allow you to project your documents across projection types while preserving the relationships across projection types. All content projected within a single projection group preserves relationships within the data across projection types. Within tables, relationships are based on a generated key and each child node retains a reference to the parent node. Across types (tables, objects and files), relationships are preserved when a single node is projected across different types. For example, consider a scenario where you

have a document containing images and text. You could project the text to tables or objects and the images to files where the tables or objects have a column/property containing the file URL.

Input shaping

Getting your data in the right shape or structure is key to effective use, be it tables or objects. The ability to shape or structure your data based on how you plan to access and use it is a key capability exposed as the **Shaper** skill within the skillset.

Projections are easier to define when you have an object in the enrichment tree that matches the schema of the projection. The updated **Shaper skill** allows you to compose an object from different nodes of the enrichment tree and parent them under a new node. The **Shaper** skill allows you to define complex types with nested objects.

When you have a new shape defined that contains all the elements you need to project out, you can now use this shape as the source for your projections or as an input to another skill.

Projection slicing

When defining a projection group, a single node in the enrichment tree can be sliced into multiple related tables or objects. Adding a projection with a source path that is a child of an existing projection will result in the child node being sliced out of the parent node and projected into the new yet related table or object. This technique allows you to define a single node in a shaper skill that can be the source for all of your projections.

Table projections

Because it makes importing easier, we recommend table projections for data exploration with Power BI. Additionally, table projections allow for changing the cardinality between table relationships.

You can project a single document in your index into multiple tables, preserving the relationships. When projecting to multiple tables, the complete shape will be projected into each table, unless a child node is the source of another table within the same group.

Defining a table projection

When defining a table projection within the `knowledgeStore` element of your skillset, start by mapping a node on the enrichment tree to the table source. Typically this node is the output of a **Shaper** skill that you added to the list of skills to produce a specific shape that you need to project into tables. The node you choose to project can be sliced to project into multiple tables. The tables definition is a list of tables that you want to project.

Each table requires three properties:

- `tableName`: The name of the table in Azure Storage.
- `generatedKeyName`: The column name for the key that uniquely identifies this row.
- `source`: The node from the enrichment tree you are sourcing your enrichments from. This node is usually the output of a shaper, but could be the output of any of the skills.

Here is an example of table projections.

```
{
  "name": "your-skillset",
  "skills": [
    ...your skills
  ],
  "cognitiveServices": {
    ... your cognitive services key info
  },

  "knowledgeStore": {
    "storageConnectionString": "an Azure storage connection string",
    "projections" : [
      {
        "tables": [
          { "tableName": "MainTable", "generatedKeyName": "SomeId", "source": "/document/EnrichedShape" },
          { "tableName": "KeyPhrases", "generatedKeyName": "KeyPhraseId", "source": "/document/EnrichedShape/*/KeyPhrases/*" },
          { "tableName": "Entities", "generatedKeyName": "EntityId", "source": "/document/EnrichedShape/*/Entities/*" }
        ]
      },
      {
        "objects": [ ]
      },
      {
        "files": [ ]
      }
    ]
  }
}
```

As demonstrated in this example, the key phrases and entities are modeled into different tables and will contain a reference back to the parent (MainTable) for each row.

Object projections

Object projections are JSON representations of the enrichment tree that can be sourced from any node. In many cases, the same **Shaper** skill that creates a table projection can be used to generate an object projection.

```
{
  "name": "your-skillset",
  "skills": [
    ...your skills
  ],
  "cognitiveServices": {
    ... your cognitive services key info
  },
  "knowledgeStore": {
    "storageConnectionString": "an Azure storage connection string",
    "projections" : [
      {
        "tables": [ ]
      },
      {
        "objects": [
          {
            "storageContainer": "hotelreviews",
            "source": "/document/hotel"
          }
        ]
      },
      {
        "files": [ ]
      }
    ]
  }
}
```

Generating an object projection requires a few object-specific attributes:

- **storageContainer**: The blob container where the objects will be saved
- **source**: The path to the node of the enrichment tree that is the root of the projection

File projection

File projections are similar to object projections and only act on the `normalized_images` collection. Similar to object projections, file projections are saved in the blob container with folder prefix of the base64 encoded value of the document ID. File projections cannot share the same container as object projections and need to be projected into a different container.

```
{
  "name": "your-skillset",
  "skills": [
    ...your skills
  ],
  "cognitiveServices": {
    ... your cognitive services key info
  },

  "knowledgeStore": {
    "storageConnectionString": "an Azure storage connection string",
    "projections" : [
      {
        "tables": [ ]
      },
      {
        "objects": [ ]
      },
      {
        "files": [
          {
            "storageContainer": "ReviewImages",
            "source": "/document/normalized_images/*"
          }
        ]
      }
    ]
  }
}
```

Projection lifecycle

Your projections have a lifecycle that is tied to the source data in your data source. As your data is updated and reindexed, your projections are updated with the results of the enrichments ensuring your projections are eventually consistent with the data in your data source. The projections inherit the delete policy you've configured for your index. Projections are not deleted when the indexer or the search service itself is deleted.

Using projections

After the indexer is run, you can read the projected data in the containers or tables you specified through projections.

For analytics, exploration in Power BI is as simple as setting Azure Table storage as the data source. You can easily create a set of visualizations on your data using the relationships within.

Alternatively, if you need to use the enriched data in a data science pipeline, you could [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.

Next steps

As a next step, create your first knowledge store using sample data and instructions.

[Create a knowledge store in REST.](#)

For a tutorial covering advanced projections concepts like slicing, inline shaping and relationships, start with [define projections in a knowledge store](#)

Define projections in a knowledge store

Incremental enrichment and caching in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

IMPORTANT

Incremental enrichment is currently in public preview. This preview version is provided without a service level agreement, and it's not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). REST API preview versions provide this feature. There is no portal or .NET SDK support at this time.

Incremental enrichment is a feature that targets [skillsets](#). It leverages Azure Storage to save the processing output emitted by an enrichment pipeline for reuse in future indexer runs. Wherever possible, the indexer reuses any cached output that is still valid.

Not only does incremental enrichment preserve your monetary investment in processing (in particular, OCR and image processing) but it also makes for a more efficient system. When structures and content are cached, an indexer can determine which skills have changed and run only those that have been modified, as well as any downstream dependent skills.

A workflow that uses incremental caching includes the following steps:

1. [Create or identify an Azure storage account](#) to store the cache.
2. [Enable incremental enrichment](#) in the indexer.
3. [Create an indexer](#) - plus a [skillset](#) - to invoke the pipeline. During processing, stages of enrichment are saved for each document in Blob storage for future use.
4. Test your code, and after making changes, use [Update Skillset](#) to modify a definition.
5. [Run Indexer](#) to invoke the pipeline, retrieving cached output for faster and more cost-effective processing.

For more information about steps and considerations when working with an existing indexer, see [Set up incremental enrichment](#).

Indexer cache

Incremental enrichment adds a cache to the enrichment pipeline. The indexer caches the results from document cracking plus the outputs of each skill for every document. When a skillset is updated, only the changed, or downstream, skills are rerun. The updated results are written to the cache and the document is updated in the search index or the knowledge store.

Physically, the cache is stored in a blob container in your Azure Storage account. The cache also uses table storage for an internal record of processing updates. All indexes within a search service may share the same storage account for the indexer cache. Each indexer is assigned a unique and immutable cache identifier to the container it is using.

Cache configuration

You'll need to set the `cache` property on the indexer to start benefitting from incremental enrichment. The following example illustrates an indexer with caching enabled. Specific parts of this configuration are described in following sections. For more information, see [Set up incremental enrichment](#).

```
{
  "name": "myIndexerName",
  "targetIndexName": "myIndex",
  "dataSourceName": "myDatasource",
  "skillsetName": "mySkillset",
  "cache" : {
    "storageConnectionString" : "Your storage account connection string",
    "enableReprocessing": true
  },
  "fieldMappings" : [],
  "outputFieldMappings": [],
  "parameters": []
}
```

Setting this property on an existing indexer will require you to reset and rerun the indexer, which will result in all documents in your data source being processed again. This step is necessary to eliminate any documents enriched by previous versions of the skillset.

Cache management

The lifecycle of the cache is managed by the indexer. 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. The cache lifecycle is also tied to the indexer lifecycle. If an indexer is deleted, the associated cache is also deleted.

While incremental enrichment is designed to detect and respond to changes with no intervention on your part, there are parameters you can use to override default behaviors:

- Prioritize new documents
- Bypass skillset checks
- Bypass data source checks
- Force skillset evaluation

Prioritize new documents

Set the `enableReprocessing` property 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. To ensure consistency across the corpus, `enableReprocessing` should be `true` most of the time, ensuring that all documents, both new and existing, are valid per the current skillset definition.

Bypass skillset evaluation

Modifying a skillset and reprocessing of that skillset typically go hand in hand. However, some changes to a skillset 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 skillset itself.

If you know that a change to the skillset is indeed superficial, you should override skillset 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 corpus.

The following example shows an Update Skillset request with the parameter:

```
PUT https://customerdemos.search.windows.net/skillsets/callcenter-text-skillset?api-version=2020-06-30-  
Preview&disableCacheReprocessingChangeDetection=true
```

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.

```
PUT https://customerdemos.search.windows.net/datasources/callcenter-ds?api-version=2020-06-30-  
Preview&ignoreResetRequirement=true
```

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, run the indexer to invoke the pipeline.

Change detection

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.

Changes that invalidate the cache

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 that would invalidate your cache:

- Change to your data source type
- Change to data source container
- Data source credentials
- Data source change detection policy
- Data sources delete detection policy
- Indexer field mappings
- 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:

- Skill in the skillset has different type. The odata type of the skill is updated
- Skill-specific parameters updated, for example the url, defaults or other parameters
- Skill outputs changes, the skill returns additional or different outputs
- Skill updates resulting in different ancestry, skill chaining has changed i.e skill inputs
- 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 reprojecting documents
- Changes to the knowledge store projections, results in reprojecting documents
- Output field mappings changed on an indexer results in reprojecting documents to the index

API reference

REST API version `2020-06-30-Preview` 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 on how to call the APIs.

- [Create Indexer \(api-version=2020-06-30-Preview\)](#)
- [Update Indexer \(api-version=2020-06-30-Preview\)](#)
- [Update Skillset \(api-version=2020-06-30\)](#) (New URI parameter on the request)
- [Reset Skills \(api-version=2020-06-30\)](#)
- Database indexers (Azure SQL, Cosmos DB). Some indexers retrieve data through queries. For queries that retrieve data, [Update Data Source](#) supports a new parameter on a request `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.

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 an existing indexer or add a cache when defining a new indexer.

[Configure caching for incremental enrichment](#)

Security in Azure Cognitive Search - overview

10/4/2020 • 10 minutes to read • [Edit Online](#)

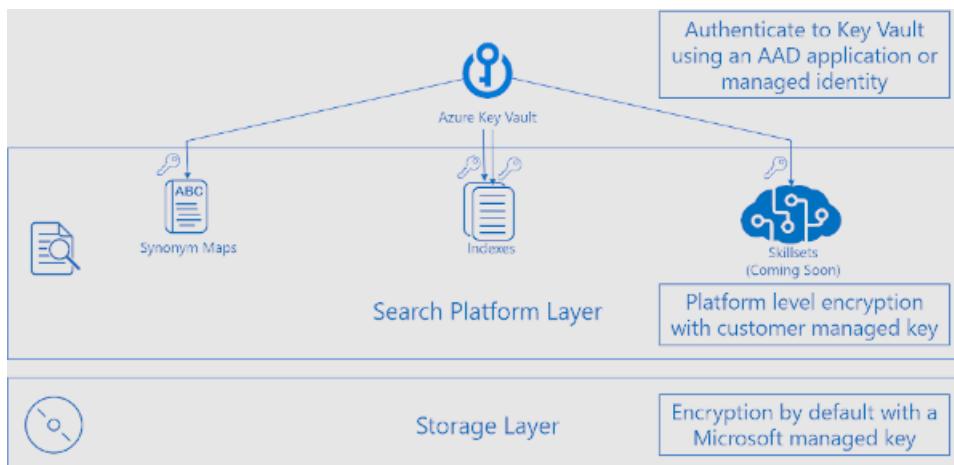
This article describes the key security features in Azure Cognitive Search that can protect content and operations.

- At the storage layer, encryption-at-rest is built in for all service-managed content saved to disk, including indexes, synonym maps, and the definitions of indexers, data sources, and skillsets. Azure Cognitive Search also supports the addition of customer-managed keys (CMK) for supplemental encryption of indexed content. For services created after August 1 2020, CMK encryption extends to data on temporary disks, for full double encryption of indexed content.
- Inbound security protects the search service endpoint at increasing levels of security: from API keys on the request, to inbound rules in the firewall, to private endpoints that fully shield your service from the public internet.
- Outbound security applies to indexers that pull content from external sources. For outbound requests, set up a managed identity to make search a trusted service when accessing data from Azure Storage, Azure SQL, Cosmos DB, or other Azure data sources. A managed identity is a substitute for credentials or access keys on the connection. Outbound security is not covered in this article. For more information about this capability, see [Connect to a data source using a managed identity](#).

Watch this fast-paced video for an overview of the security architecture and each feature category.

Encrypted transmissions and storage

In Azure Cognitive Search, encryption starts with connections and transmissions, and extends to content stored on disk. 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) are not supported.



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. Those services have their own levels of encryption support separate from Azure Cognitive Search.

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)
server-side encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in all regions, for content created after January 2019.	Content (indexes and synonym maps) on data disks
server-side double encryption	customer-managed keys	Azure Key Vault	Available on billable tiers, in selected regions, on search services after August 1 2020.	Content (indexes and synonym maps) on data disks and temporary disks

Service-managed keys

Service-managed encryption is a Microsoft-internal operation, based on [Azure Storage Service Encryption](#), using 256-bit [AES encryption](#). It occurs automatically on all indexing, including on incremental updates to indexes that are not fully encrypted (created before January 2018).

Customer-managed keys (CMK)

Customer-managed keys require an additional billable service, Azure Key Vault, which can be in a different region, but under the same subscription, as Azure Cognitive Search. 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% in query times, although actual performance will vary depending on the index definition and types of queries. Because of this 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](#).

Double encryption

In Azure Cognitive Search, double encryption is an extension of CMK. It is understood to be two-fold encryption (once by CMK, and again by service-managed keys), and comprehensive in scope, encompassing long term storage that is written to a data disk, and short term storage written to temporary disks. The difference between CMK before August 1 2020 and after, and what makes CMK a double encryption feature in Azure Cognitive Search, is the additional encryption of data-at-rest on temporary disks.

Double encryption is currently available on new services that are created in these regions after August 1:

- West US 2
- East US
- South Central US
- US Gov Virginia
- US Gov Arizona

Inbound security and endpoint protection

Inbound security features protect the search service endpoint through increasing levels of security and complexity. First, all requests require an API key for authenticated access. Second, you can optionally set firewall rules that limit access to specific IP addresses. For advanced protection, a third option is to enable Azure Private Link to shield your service endpoint from all internet traffic.

Public access using API keys

By default, a search service is accessed through the public cloud, using key-based authentication for admin or

query access to the search service endpoint. An api-key is a string composed of randomly generated numbers and letters. The type of key (admin or query) determines the level of access. Submission of a valid key is considered proof the request originates from a trusted entity.

There are two levels of access to your search service, enabled by the following API keys:

- Admin key (allows read-write access for [create-read-update-delete](#) operations on the search service)
- Query key (allows read-only access to the documents collection of an index)

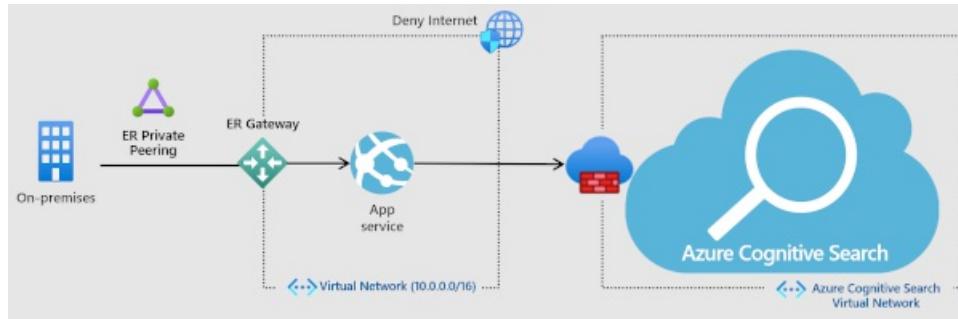
Admin keys are created when the service is provisioned. There are two admin keys, designated as *primary* and *secondary* to keep them straight, but in fact they are interchangeable. Each service has two admin keys so that you can roll one over without losing access to your service. You can [regenerate admin key](#) periodically per Azure security best practices, but you cannot add to the total admin key count. There are a maximum of two admin keys per search service.

Query keys are created as-needed and are designed for client applications that issue queries. You can create up to 50 query keys. In application code, you specify the search URL and a query api-key to allow read-only access to the documents collection of a specific index. Together, the endpoint, an api-key for read-only access, and a target index define the scope and access level of the connection from your client application.

Authentication is required on each request, where each request is composed of a mandatory key, an operation, and an object. When chained together, the two permission levels (full or read-only) plus the context (for example, a query operation on an index) are sufficient for providing full-spectrum security on service operations. For more information about keys, see [Create and manage api-keys](#).

IP-restricted access

To further control access to your search service, you can create inbound firewall rules that allow access to 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.



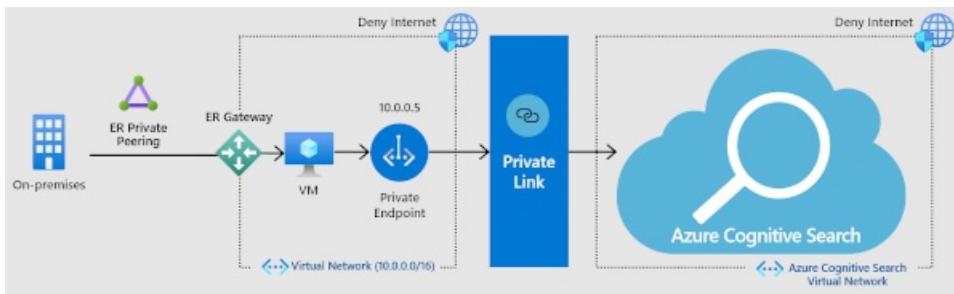
You can use the portal to [configure inbound access](#).

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.

Private endpoint (no Internet traffic)

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 additional services is an added cost so be sure you have a clear understanding of the benefits before diving in. or more information about costs, see the [pricing page](#). For more information about how these components work together, watch the video at the top of this article. 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](#).

Index access

In Azure Cognitive Search, an individual index is not a securable object. Instead, access to an index is determined at the service layer (read or write access to the service), along with the context of an operation.

For end-user access, you can structure query requests to connect using a query key, which makes any request read-only, and include the specific index used by your app. In a query request, there is no concept of joining indexes or accessing multiple indexes simultaneously so all requests target a single index by definition. As such, construction of the query request itself (a key plus a single target index) defines the security boundary.

Administrator and developer access to indexes is undifferentiated: both need write access to create, delete, and update objects managed by the service. 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 remedy for reversing an unwanted index deletion or modification. Azure Cognitive Search has failover within the cluster to ensure availability, but it does not store or execute your proprietary code used to create or load indexes.

For multitenancy solutions requiring security boundaries at the index level, such solutions typically include a middle tier, which customers use to handle index isolation. For more information about the multitenant use case, see [Design patterns for multitenant SaaS applications and Azure Cognitive Search](#).

User access

How a user accesses an index and other objects is determined by the type of API key on the request. Most developers create and assign [query keys](#) for client-side search requests. A query key grants read-only access to searchable content within the index.

If you require granular, per-user control over search results, you can build security filters on your queries, returning documents associated with a given security identity. Instead of predefined roles and role assignments, identity-based access control is implemented as a *filter* that trims search results of documents and content based on identities. 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.

APPROACH	DESCRIPTION
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.

Administrative rights

Azure role-based access control (Azure RBAC) is an authorization system built on [Azure Resource Manager](#) for provisioning of Azure resources. In Azure Cognitive Search, Resource Manager is used to create or delete the service, manage API keys, and scale the service. As such, Azure role assignments will determine who can perform those tasks, regardless of whether they are using the [portal](#), [PowerShell](#), or the [Management REST APIs](#).

In contrast, admin rights over content hosted on the service, such as the ability to create or delete an index, is conferred through API keys as described in the [previous section](#).

TIP

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](#).

Certifications and compliance

Azure Cognitive Search has been certified compliant for multiple 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 [Azure Security Benchmark](#). Azure 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 11 security controls, including [Network Security](#), [Logging and Monitoring](#), and [Data Protection](#) to name a few.

Azure Policy is a capability built into Azure that helps you manage compliance for multiple standards, including those of Azure Security Benchmark. For well-known benchmarks, Azure Policy provides built-in definitions that provide both criteria as well as an actionable response that addresses non-compliance.

For Azure Cognitive Search, there is currently one built-in definition. It is for diagnostic logging. With this built-in, you can assign a policy that identifies any search service that is missing diagnostic logging, and then turns it on. For more information, see [Azure Policy Regulatory Compliance controls for Azure Cognitive Search](#).

See also

- [Azure security fundamentals](#)
- [Azure Security](#)
- [Azure Security Center](#)

Azure Policy Regulatory Compliance controls for Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

If you are using [Azure Policy](#) to enforce the recommendations in [Azure 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 [diagnostic 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 below is associated with one or more [Azure Policy](#) definitions. These policies may help you [assess compliance](#) with the control; however, there often is not 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 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 may 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 Monitoring	2.3	Enable audit logging for Azure resources	Diagnostic logs in Search services should be enabled	3.0.0

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](#).

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)
Audit Logging	1208.09aa3System.1 - 09.aa	Audit logs are maintained for management activities, system and application startup/shutdown/errors, file changes, and security policy changes.	Diagnostic logs in Search services should be enabled	3.0.0

Next steps

- Learn more about [Azure Policy Regulatory Compliance](#).
- See the built-ins on the [Azure Policy GitHub repo](#).

Indexer access to data sources using Azure network security features

10/4/2020 • 7 minutes to read • [Edit Online](#)

Azure Cognitive Search indexers can make outbound calls to various Azure resources during execution. This article explains the concepts behind indexer access to resources when those resources are protected by IP firewalls, private endpoints, and other network-level security mechanisms. The possible 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, tables, ADLS Gen 2)	Data source
Azure Storage (blobs, tables)	Skillsets (caching enriched documents, and storing knowledge store projections)
Azure Cosmos DB (various APIs)	Data source
Azure SQL Database	Data source
SQL server on Azure IaaS VMs	Data source
SQL managed instances	Data source
Azure Functions	Host for custom web api skills
Cognitive Services	Attached to skillset that will be used to bill enrichment beyond the 20 free documents limit

NOTE

The cognitive service resource attached to a skillset is used for billing, based on the enrichments performed and written into the search index. It is not used for accessing the Cognitive Services APIs. Access from an indexer's enrichment pipeline to Cognitive Services APIs occurs via a secure communication channel, where data is strongly encrypted in transit and is never stored at rest.

Customers can secure these resources via several network isolation mechanisms offered by Azure. With the exception of cognitive service resource, indexers have limited ability to access all other resources even if they are network isolated outlined in the table below.

RESOURCE	IP RESTRICTION	PRIVATE ENDPOINT
Azure storage (blobs, tables, ADLS Gen 2)	Supported only if the storage account and search service are in different regions	Supported
Azure Cosmos DB - SQL API	Supported	Supported

RESOURCE	IP RESTRICTION	PRIVATE ENDPOINT
Azure Cosmos DB - Cassandra, Mongo, and Gremlin API	Supported	Unsupported
Azure SQL Database	Supported	Supported
SQL server on Azure IaaS VMs	Supported	N/A
SQL managed instances	Supported	N/A
Azure Functions	Supported	Supported, only for certain SKUs of Azure functions

NOTE

In addition to the options listed above, for network secured Azure storage accounts, customers can leverage the fact that Azure Cognitive Search is a [trusted Microsoft service](#). This means that a specific search service can bypass virtual network or IP restrictions on the storage account and can access data in the storage account, if the appropriate role based access control is enabled on the storage account. Details are available in the [how to guide](#). This option can be utilized instead of the IP restriction route, in case either the storage account or the search service cannot be moved to a different region.

When choosing which secure access mechanism that an indexer should use, consider the following constraints:

- [Service endpoints](#) will not be supported for any Azure resource.
- A search service cannot be provisioned into a specific virtual network - this functionality will not be offered by Azure Cognitive Search.
- When indexers utilize (outbound) private endpoints to access resources, additional [private link charges](#) may apply.

Indexer execution environment

Azure Cognitive Search indexers are capable of efficiently extracting content from data sources, adding enrichments to the extracted content, optionally generating projections before writing the results to the search index. Depending on the number of responsibilities assigned to an indexer, it can run in one of two environments:

- An environment private to a specific search service. Indexers running in such environments share resources with other workloads (such as other customer initiated indexing or querying workload). Typically only indexers that do not require much resources (for example, do not use a skillset) run in this environment.
- A multi-tenant environment that hosts indexers that are resource hungry, such as ones with a skillset. Resource hungry resources run on this environment to offer optimal performance whilst ensuring that the search service resources are available for other workloads. This multi-tenant environment is managed and secured by Azure Cognitive Search, at no extra cost to the customer.

For any given indexer run, Azure Cognitive Search determines the best environment in which to run the indexer.

Granting access to indexer IP ranges

If the resource that your indexer is trying to access is restricted to only a certain set of IP ranges, then you need to expand the set to include the possible IP ranges from which an indexer request can originate. As stated above, there are two possible environments in which indexers run and from which access requests can originate. You will need to add the IP addresses of **both** environments for indexer access to work.

- To obtain the IP address of the search service specific private environment, `nslookup` (or `ping`) the fully

qualified domain name (FQDN) of your search service. The FQDN of a search service in the public cloud, for example, would be <service-name>.search.windows.net . This information is available on the Azure portal.

- The IP addresses of the multi-tenant environments are available via the AzureCognitiveSearch service tag. Azure service tags have a published range of IP addresses for each service - this is available via a [discovery API \(preview\)](#) or a [downloadable JSON file](#). In either case, IP ranges are broken down by region - you can pick only the IP ranges assigned for the region in which your search service is provisioned.

For certain data sources, the service tag itself can be used directly instead of enumerating the list of IP ranges (the IP address of the search service still needs to be used explicitly). These data sources restrict access by means of setting up a [Network Security Group rule](#), which natively support adding a service tag, unlike IP rules such as the ones offered by Azure Storage, CosmosDB, Azure SQL etc. The data sources that support the ability to utilize the AzureCognitiveSearch service tag directly in addition to search service IP address are:

- [SQL server on IaaS VMs](#)
- [SQL managed instances](#)

Details are described in the [how to guide](#).

Granting access via private endpoints

Indexers can utilize [private endpoints](#) to access resources, access to which are locked down either to select virtual networks or do not have any public access enabled. This functionality is only available for paid services, with limits on the number of private endpoints that can be created. Details about the limits are documented in the [Azure Search limits page](#).

Step 1: Create a private endpoint to the secure resource

Customers should call the search management operation, [Create or Update shared private link resource API](#) in order to create a private endpoint connection to their secure resource (for example, a storage account). Traffic that goes over this (outbound) private endpoint connection will originate only from the virtual network that's in the search service specific "private" indexer execution environment.

Azure Cognitive Search will validate that callers of this API have permissions to approve private endpoint connection requests to the secure resource. For example, if you request a private endpoint connection to a storage account that you do not have access to, this call will be rejected.

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. The customer is then expected to locate this request on their secure resource and "Approve" it. Typically, this can be done either via the Portal or via the [REST API](#).

Step 3: Force indexers to run in the "private" environment

An approved private endpoint allows outgoing calls from the search service to a resource that has some form of network level access restrictions (for example a storage account data source that is configured to only be accessed from certain virtual networks) to succeed. This means any indexer that is able to reach out to such a data source over the private endpoint will succeed. If the private endpoint is not approved, or if the indexer does not utilize the private endpoint connection then the indexer run will end up in transientFailure .

To enable indexers to access resources via private endpoint connections, it is mandatory to set the executionEnvironment of the indexer to "Private" to ensure that all indexer runs will be able to utilize the private endpoint. This is because private endpoints are provisioned within the private search service-specific environment.

```
{  
    "name" : "myindexer",  
    ... other indexer properties  
    "parameters" : {  
        ... other parameters  
        "configuration" : {  
            ... other configuration properties  
            "executionEnvironment": "Private"  
        }  
    }  
}
```

These steps are described in greater detail in the [how to guide](#). Once you have an approved private endpoint to a resource, indexers that are set to be *private* attempt to obtain access via the private endpoint connection.

Limits

To ensure optimal performance and stability of the search service, restrictions are imposed (by search service SKU) on the following dimensions:

- The kinds of indexers that can be set to be *private*.
- The number of shared private link resources that can be created.
- The number of distinct resource types for which shared private link resources can be created.

These limits are documented in [service limits](#).

Azure security baseline for Azure Cognitive Search

10/4/2020 • 29 minutes to read • [Edit Online](#)

This security baseline applies guidance from the [Azure Security Benchmark version 1.0](#) to Azure Cognitive Search. The Azure 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 Azure Security Benchmark and the related guidance applicable to Azure Cognitive Search. Controls not applicable to Azure Cognitive Search, or the customer have been excluded.

To see how Azure Cognitive Search completely maps to the Azure Security Benchmark, see the [full Azure Cognitive Search security baseline mapping file](#).

Network security

For more information, see the [Azure Security Benchmark: Network security](#).

1.1: Protect Azure resources within virtual networks

Guidance: Ensure that all Microsoft Azure Virtual Network subnet deployments have a network security group applied with rules to implement a "least privileged" access scheme. Allow access only to your application's trusted ports and IP address ranges. Deploy Azure Cognitive Search with an Azure private endpoint, where feasible, to enable private access to your services from your virtual network.

Cognitive Search also supports additional network security functionality for managing network access control lists. Configure your search service to only allow communication with trusted sources by restricting access from specific public IP address ranges using its firewall capability.

- [How to configure Private Endpoints for Azure Cognitive Search](#)
- [How to configure the Azure Cognitive Search firewall](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.2: Monitor and log the configuration and traffic of virtual networks, subnets, and NICs

Guidance: Cognitive Search cannot be deployed directly into a virtual network. However, if your client application or data sources are in a virtual network, you can monitor and log traffic for those in-network components, including requests sent to a search service in the cloud. Standard recommendations include enabling a network security group flow log and sending logs to either Azure Storage or a Log Analytics workspace. You could optionally use Traffic Analytics for insights into traffic patterns.

- [How to enable network security group flow logs](#)
- [How to enable and use Traffic Analytics](#)
- [Understand network security provided by Azure Security Center](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

1.3: Protect critical web applications

Guidance: Not applicable to Cognitive Search. This recommendation is intended for web applications running on Azure App Service or compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

1.4: Deny communications with known malicious IP addresses

Guidance: Cognitive Search does not provide a specific feature to combat a distributed denial-of-service attack, but you can enable DDoS Protection Standard on the virtual networks associated with your Cognitive Search service for general protection.

- [How to configure DDoS protection](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.5: Record network packets

Guidance: Enable network security group flow logs for the network security groups protecting Azure Virtual Machines (VM) that will be connecting to your Cognitive Search service. Send logs into an Azure Storage account for traffic audit.

Enable Network Watcher packet capture if required for investigating anomalous activity.

- [How to Enable NSG Flow Logs](#)
- [How to enable Network Watcher](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.6: Deploy network-based intrusion detection/intrusion prevention systems (IDS/IPS)

Guidance: Cognitive Search does not support network intrusion detection, but as an intrusion mitigation, you can configure firewall rules to specify the IP addresses accepted by the Cognitive Search service. Configure a private endpoint to keep search traffic away from the public internet.

- [How to configure customer-managed keys for data encryption](#)
- [How to get customer-managed key information from indexes and synonym maps](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.7: Manage traffic to web applications

Guidance: Not applicable to Cognitive Search. This recommendation is intended for web applications running on Azure App Service or compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

1.8: Minimize complexity and administrative overhead of network security rules

Guidance: Use service tags, if you are leveraging indexers and skillsets in Cognitive Search, to represent a range of IP addresses that have permission to connect to external resources.

Allow or deny traffic to resources by specifying the service tag name (for example, AzureCognitiveSearch) in the appropriate source or destination field of a rule.

- [Virtual network service tags](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.9: Maintain standard security configurations for network devices

Guidance: Cognitive Search does not have or depend on network resources by design. Client apps and data sources related to your search application might be on a virtual network, but the search service is not itself deployed in the network.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

1.10: Document traffic configuration rules

Guidance: You can configure Cognitive Search with an Azure private endpoint to integrate your search service with a virtual network. Use resource tags for network security groups and other resources related to network security and traffic flow. For individual network security group rules, use the "Description" field to document the rules that allow traffic to/from a network.

Use any of the built-in Azure Policy definitions related to tagging, such as "Require tag and its value" effects, to ensure that all resources are created with tags and to notify you of existing untagged resources.

You can use Azure PowerShell or Azure CLI to look-up or perform actions on resources based on their tags.

- [How to create a private endpoint for Cognitive Search](#)
- [How to create and use tags](#)
- [How to create an Azure Virtual Network](#)
- [How to filter network traffic with network security group rules](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

1.11: Use automated tools to monitor network resource configurations and detect changes

Guidance: Cognitive Search does not have or depend on any networking components, so the configurations of these resources cannot be monitored.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Logging and monitoring

For more information, see the [Azure Security Benchmark: Logging and monitoring](#).

2.1: Use approved time synchronization sources

Guidance: Cognitive Search does not support configuring your own time synchronization sources. The search service relies on Microsoft time synchronization sources, and is not exposed to customers for configuration.

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

2.2: Configure central security log management

Guidance: Ingest logs related to Cognitive Search via Azure Monitor to aggregate security data generated by endpoint devices, network resources, and other security systems. In Azure Monitor, use Log Analytics workspaces to query and perform analytics, and use Azure Storage accounts for long term and archival storage. Alternatively, you can enable and on-board this data to Azure Sentinel or a third-party SIEM.

- [How to get started with Azure Monitor and third-party SIEM integration](#)
- [How to collect platform logs and metrics with Azure Monitor](#)
- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.3: Enable audit logging for Azure resources

Guidance: Diagnostic and operational logs provide insight into the detailed operations of Cognitive Search and are useful for monitoring the service and for workloads that access your service. To capture diagnostic data, enable logging by specifying where logging information is stored.

- [How to collect and analyze log data for Azure Cognitive Search](#)
- [How to collect platform logs and metrics with Azure Monitor](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

2.4: Collect security logs from operating systems

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.5: Configure security log storage retention

Guidance: Historical data that feeds into diagnostic metrics is preserved by Cognitive Search for 30 days by default. For longer retention, be sure to enable the setting that specifies a storage option for persisting logged events and metrics.

In Azure Monitor, set your Log Analytics workspace retention period according to your organization's compliance regulations. Use Azure Storage accounts for long-term and archival storage.

- [Change the data retention period in Log Analytics](#)
- [How to configure retention policy for Azure Storage account logs](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

2.6: Monitor and review Logs

Guidance: Analyze and monitor logs from your Cognitive Search service for anomalous behavior. Use Azure Monitor's Log Analytics to review logs and perform queries on log data. Alternatively, you may enable and onboard data to Azure Sentinel or a third party SIEM.

- [How to collect and analyze log data for Cognitive Search](#)
- [How to visualize search log data in Power BI](#)
- [How to onboard Azure Sentinel](#)
- [Learn about Log Analytics](#)
- [How to perform custom queries in Azure Monitor](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

2.7: Enable alerts for anomalous activities

Guidance: Use Security Center with Log Analytics workspace for monitoring and alerting on anomalous activity found in security logs and events. Alternatively, you can enable and on-board data to Azure Sentinel.

- [How to onboard Azure Sentinel](#)
- [How to manage alerts in Azure Security Center](#)
- [How to alert on log analytics log data](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

2.8: Centralize anti-malware logging

Guidance: Not applicable to Cognitive Search. Microsoft manages the anti-malware solution for the underlying platform.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.9: Enable DNS query logging

Guidance: Not applicable to Cognitive Search. It does not produce or consume DNS logs.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

2.10: Enable command-line audit logging

Guidance: Not applicable to Cognitive Search. Command-line auditing is not available for Cognitive Search.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Identity and access control

For more information, see the [Azure Security Benchmark: Identity and access control](#).

3.1: Maintain an inventory of administrative accounts

Guidance: Azure role-based access control (Azure RBAC) allows you to manage access to Azure resources through role assignments. You can assign these roles to users, groups service principals and managed identities. There are pre-defined built-in roles for certain resources, and these roles can be inventoried or queried through tools such as Azure CLI, Azure PowerShell or the Azure portal.

Cognitive Search roles are associated with permissions that support service level management tasks. These roles do not grant access to the service endpoint. Access to operations against the endpoint, (such as index management, index population, and queries on search data), use API keys to authenticate the request.

- [Set roles for administrative access to Azure Cognitive Search](#)
- [Create and manage api-keys for an Azure Cognitive Search service](#)
- [How to get a directory role in Azure AD with PowerShell](#)
- [How to get members of a directory role in Azure AD with PowerShell](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.2: Change default passwords where applicable

Guidance: Not Applicable to Cognitive Search. It does not have a concept of default passwords.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

3.3: Use dedicated administrative accounts

Guidance: Cognitive Search does not have the concept of any local-level or Azure Active Directory (Azure AD) administrator accounts that can be used to manage indexes and operations.

Use the Azure AD built-in roles which must be explicitly assigned for management operations. Invoke the Azure AD PowerShell module to perform ad-hoc queries to discover accounts that are members of administrative groups.

- [How to use roles for administrative access in Cognitive Search](#)
- [How to get a directory role in Azure AD with PowerShell](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.4: Use single sign-on (SSO) with Azure Active Directory

Guidance: Use SSO authentication with Azure Active Directory (Azure AD) to access search service information for management operations supported through Azure Resource Manager.

Establish a process to reduce the number of identities and credentials by enabling SSO for the service with your organization's pre-existing identities.

- [Understand SSO with Azure AD](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.5: Use multi-factor authentication for all Azure Active Directory based access

Guidance: Enable Azure Active Directory's (Azure AD) Multi-Factor Authentication (MFA) feature and follow Security Center's Identity and Access recommendations.

- [How to enable MFA in Azure](#)
- [How to monitor identity and access within Azure Security Center](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.6: Use dedicated machines (Privileged Access Workstations) for all administrative tasks

Guidance: Use a Privileged Access Workstation (PAW) with Multi-Factor Authentication (MFA) configured to log into and access Azure resources.

- [Understand secure, Azure-managed workstations](#)
- [How to enable Azure AD MFA](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

3.7: Log and alert on suspicious activities from administrative accounts

Guidance: Use Azure Active Directory (Azure AD) security reports and monitoring to detect when suspicious or unsafe activity occurs in the environment. Use Security Center to monitor identity and access activity.

- [How to identify Azure AD users flagged for risky activity](#)
- [How to monitor users' identity and access activity in Azure Security Center](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

3.8: Manage Azure resources only from approved locations

Guidance: Not applicable to Cognitive Search. It does not support using approved location as condition for access.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

3.9: Use Azure Active Directory

Guidance: Use Azure Active Directory (Azure AD) as the central authentication and authorization system for service level management tasks in Azure Cognitive Search. Azure AD identities do not grant access to the search service endpoint. Access to operations such as index management, index population, and queries on search data are available via API keys.

- [How to create and configure an Azure AD instance](#)
- [Create and manage api-keys for an Azure Cognitive Search service](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.10: Regularly review and reconcile user access

Guidance: Azure Active Directory (Azure AD) provides logs to help discover stale accounts. Use Azure AD's Identity and access reviews to efficiently manage group memberships, access to enterprise applications, and role assignments. User access can be reviewed on a regular basis to make sure only the right users have continued access.

Review diagnostic logs from Cognitive Search for activity in the search service endpoint such as index management, index population, and queries.

- [Understand Azure AD reporting](#)
- [How to use Azure AD identity and access reviews](#)
- [Monitor operations and activity of Azure Cognitive Search](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.11: Monitor attempts to access deactivated credentials

Guidance: Access to Azure Active Directory (Azure AD) sign-in activity, audit, and risk event log sources, allow you to integrate with any SIEM or monitoring tool.

Streamline this process by creating diagnostic settings for Azure AD user accounts and sending the audit logs and sign-in logs to a Log Analytics workspace. Configure desired alerts within Log Analytics workspace.

- [How to integrate Azure activity logs with Azure Monitor](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.12: Alert on account login behavior deviation

Guidance: Use Azure Active Directory (Azure AD) Identity Protection features to configure automated responses to detected suspicious actions related to user identities. Ingest data into Azure Sentinel for further investigation, as required.

- [How to view Azure AD risky sign-ins](#)
- [How to configure and enable Identity Protection risk policies](#)
- [How to onboard Azure Sentinel](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

3.13: Provide Microsoft with access to relevant customer data during support scenarios

Guidance: Not applicable to Cognitive Search. Customer Lockbox does not support Cognitive Search.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Data protection

For more information, see the [Azure Security Benchmark: Data protection](#).

4.1: Maintain an inventory of sensitive Information

Guidance: Use tags to assist in tracking Azure resources that store or process sensitive information.

Azure Security Center monitoring: Currently not available

Responsibility: Customer

4.2: Isolate systems storing or processing sensitive information

Guidance: Implement separate subscriptions and/or management groups for development, test, and production. Resources should be separated by virtual network/subnet, tagged appropriately, and secured within a network security group or Azure Firewall. Resources storing or processing sensitive data should be isolated. Use Private Link to configure a private endpoint to Cognitive Search.

- [How to create additional Azure subscriptions](#)
- [How to create and use tags](#)
- [How to create a private endpoint for Cognitive Search](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

4.3: Monitor and block unauthorized transfer of sensitive information

Guidance: Use a third-party solution from Azure Marketplace in network perimeters to monitor for unauthorized transfer of sensitive information and block such transfers while alerting information security professionals.

Microsoft manages the underlying platform and treats all customer content as sensitive and guards against customer data loss and exposure. To ensure customer data within Azure remains secure, Microsoft has implemented and maintains a suite of robust data protection controls and capabilities.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.4: Encrypt all sensitive information in transit

Guidance: Cognitive Search encrypts data in transit with Transport Layer Security 1.2 and enforces encryption (SSL/TLS) at all times for all connections. This ensures all data is encrypted "in transit" between the client and the service.

- [Understand encryption in transit with Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

4.5: Use an active discovery tool to identify sensitive data

Guidance: Data identification, classification, and loss prevention features are not yet available for Cognitive Search. Implement a third-party solution if necessary for compliance purposes.

Microsoft manages the underlying platform and treats all customer content as sensitive and guards against customer data loss and exposure. To ensure customer data within Azure remains secure, Microsoft has implemented and maintains a suite of robust data protection controls and capabilities.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.6: Use Azure RBAC to manage access to resources

Guidance: For service administration, use Azure role-based access control (Azure RBAC) to manage access to keys and configuration. For content operations, such as indexing and queries, Cognitive Search uses keys instead of an identity-based access control model. Use Azure RBAC to control access to keys.

- [How to configure RBAC in Azure](#)
- [How to use roles for administrative access to Cognitive Search](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

4.7: Use host-based data loss prevention to enforce access control

Guidance: Not applicable to Cognitive Search. This guideline is intended for compute resources.

Microsoft manages the underlying infrastructure for Cognitive Search and has implemented strict controls to prevent the loss or exposure of customer data.

- [Understand customer data protection in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

4.8: Encrypt sensitive information at rest

Guidance: Cognitive Search automatically encrypts indexed content at rest with Microsoft-managed keys. If more protection is needed, you can supplement default encryption with a second encryption layer using keys that you create and manage in Azure Key Vault.

- [Configure customer-managed keys for data encryption in Azure Cognitive Search](#)

- [Understand encryption at rest in Azure](#)

Azure Security Center monitoring: Currently not available

Responsibility: Shared

4.9: Log and alert on changes to critical Azure resources

Guidance: Use Azure Monitor with the Azure Activity Log to create alerts for when changes take place to production instances of Cognitive Search and other critical or related resources.

- [How to create alerts for Azure Activity Log events](#)
- [How to create alerts for Cognitive Search activities](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

Vulnerability management

For more information, see the [Azure Security Benchmark: Vulnerability management](#).

5.1: Run automated vulnerability scanning tools

Guidance: Currently not available to Cognitive Search. For clusters that store search service content, Microsoft is responsible for vulnerability management of those clusters.

Azure Security Center monitoring: Currently not available

Responsibility: Microsoft

5.2: Deploy automated operating system patch management solution

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.3: Deploy an automated patch management solution for third-party software titles

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

5.4: Compare back-to-back vulnerability scans

Guidance: Not applicable to Cognitive Search. Microsoft performs vulnerability management on the underlying systems that support Cognitive Search services.

Azure Security Center monitoring: Not applicable

Responsibility: Microsoft

5.5: Use a risk-rating process to prioritize the remediation of discovered vulnerabilities

Guidance: Not applicable to Cognitive Search. It does not have any standard risk-rating or scoring system in place for vulnerability scan results.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Inventory and asset management

For more information, see the [Azure Security Benchmark: Inventory and asset management](#).

6.1: Use automated asset discovery solution

Guidance: Use Azure Resource Graph to query for and discover all resources (such as compute, storage, network, ports, protocols, and so on) in your subscriptions.

Ensure appropriate (read) permissions in your tenant and enumerate all Azure subscriptions as well as resources in your subscriptions.

- [How to create queries with Azure Resource Graph Explorer](#)
- [How to view your Azure subscriptions](#)
- [Understand Azure RBAC](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.2: Maintain asset metadata

Guidance: Apply tags to Azure resources with metadata to logically organize them into a taxonomy.

- [How to create and use tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.3: Delete unauthorized Azure resources

Guidance: Use tagging, management groups, and separate subscriptions where appropriate, to organize and track assets. Reconcile inventory on a regular basis and ensure unauthorized resources are deleted from the subscription in a timely manner.

- [How to create additional Azure subscriptions](#)
- [How to create Management Groups](#)
- [How to create and use Tags](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.4: Define and maintain an inventory of approved Azure resources

Guidance: Define a list of approved Azure resources related to indexing and skillset processing in Cognitive Search.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.5: Monitor for unapproved Azure resources

Guidance: It is recommended that you define an inventory of Azure resources which have been approved for usage as per your organizational policies and standards prior, then monitor for unapproved Azure resources with Azure Policy, or Azure Resource Graph.

- [How to configure and manage Azure Policy](#)
- [How to create queries with Azure Graph](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

6.6: Monitor for unapproved software applications within compute resources

Guidance: Not applicable to Cognitive Search. This guidance is intended for compute resources.

It is recommended that you have an inventory of software applications which have been deemed approved as per your organizational policies and security standards, and monitor for any unapproved software titles installed on your Azure compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.7: Remove unapproved Azure resources and software applications

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.8: Use only approved applications

Guidance: Not applicable to Cognitive Search. It does not expose any compute resources or allows installation of software applications on any of its resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.9: Use only approved Azure services

Guidance: Use Azure Policy to place restrictions on the type of resources that can be created in customer subscriptions using the following built-in policy definitions:

- Not allowed resource types
- Allowed resource types

Use Azure Resource Graph to query or discover resources within your subscription(s). Ensure that all Azure resources present in the environment are approved.

- [How to configure and manage Azure Policy](#)
- [How to deny a specific resource type with Azure Policy](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

6.10: Maintain an inventory of approved software titles

Guidance: Not applicable to Cognitive Search. This recommendation is intended for applications running on compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.11: Limit users' ability to interact with Azure Resource Manager

Guidance: For service management, use Azure Conditional Access to limit users' ability to interact with Azure Resource Manager by configuring "Block access" for the "Microsoft Azure Management" App.

Control access to the keys used to authenticate requests for all other operations, particularly those related to

content with Cognitive Search.

Azure Security Center monitoring: Currently not available

Responsibility: Customer

6.12: Limit users' ability to execute scripts in compute resources

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

6.13: Physically or logically segregate high risk applications

Guidance: Not applicable to Cognitive Search. This recommendation is intended for web applications running on Azure App Service or compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Secure configuration

For more information, see the [Azure Security Benchmark: Secure configuration](#).

7.1: Establish secure configurations for all Azure resources

Guidance: Use Azure Policy aliases in the "Microsoft.Search" namespace to create custom policies to audit or enforce the configuration of your Azure Cognitive Search resources. You may also use built-in Azure Policy definitions for Cognitive Search services such as:

- Enable audit logging for Azure resources

Azure Resource Manager has the ability to export the template in JavaScript Object Notation (JSON), which should be reviewed to ensure that the configurations meet the security requirements for your organization.

You can also use the recommendations from Azure Security Center as a secure configuration baseline for your Azure resources.

- [Azure Policy Regulatory Compliance controls for Azure Cognitive Search](#)
- [How to view available Azure Policy aliases](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

7.2: Establish secure operating system configurations

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.3: Maintain secure Azure resource configurations

Guidance: Use Azure Policy [deny] and [deploy if not exist] effects, to enforce secure settings across your Cognitive Search service resources.

Azure Resource Manager templates can be used to maintain the security configuration of your Azure resources required by your organization.

- [Understand Azure Policy effects](#)

- [Azure Policy Regulatory Compliance controls for Azure Cognitive Search](#)
- [Create and manage policies to enforce compliance](#)
- [Azure Resource Manager templates overview](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.4: Maintain secure operating system configurations

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.5: Securely store configuration of Azure resources

Guidance: If using custom Azure Policy definitions, use Azure DevOps or Azure Repos to securely store and manage your code.

- [How to store code in Azure DevOps](#)
- [Azure Repos documentation](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.6: Securely store custom operating system images

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.7: Deploy configuration management tools for Azure resources

Guidance: Define and implement standard security configurations for your Cognitive Search service resources using Azure Policy.

Use aliases to create custom policies to audit or enforce the network configurations. You can also make use of built-in policy definitions related to your specific resources.

Additionally, you can use Azure Automation to deploy configuration changes and manage policy exceptions.

- [Azure Policy Regulatory Compliance controls for Azure Cognitive Search](#)
- [How to configure and manage Azure Policy](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

7.8: Deploy configuration management tools for operating systems

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.9: Implement automated configuration monitoring for Azure resources

Guidance: Use Security Center to perform baseline scans of your Cognitive Search service resources. Additionally,

use Azure Policy to alert and audit your resource configurations.

- [How to remediate recommendations in Azure Security Center](#)
- [Azure Policy Regulatory Compliance controls for Azure Cognitive Search](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

7.10: Implement automated configuration monitoring for operating systems

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

7.11: Manage Azure secrets securely

Guidance: Use Azure Managed Identities in conjunction with Azure Key Vault to simplify secret management for your cloud applications.

- [How to use managed identities for Azure resources](#)
- [How to create a Key Vault](#)
- [How to provide Key Vault authentication with a managed identity](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

7.12: Manage identities securely and automatically

Guidance: Use an Azure Managed Identity to give Cognitive Search access to other Azure services such as Key Vault and indexer data sources using an automatically-managed identity in Azure Active Directory (Azure AD). Managed identities allow you to authenticate to any service that supports Azure AD authentication, including Azure Key Vault, without any credentials in your code.

- [Set up an indexer connection to a data source using a managed identity](#)
- [Configure customer-managed keys for data encryption using a managed identity](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

7.13: Eliminate unintended credential exposure

Guidance: Not applicable to Cognitive Search. It does not host code and does not have any credentials to identify.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

Malware defense

For more information, see the [Azure Security Benchmark: Malware defense](#).

8.1: Use centrally managed antimalware software

Guidance: Not applicable to Cognitive Search. This recommendation is intended for compute resources.

Microsoft anti-malware is enabled on the underlying host that supports Azure services (for example, Azure Cognitive Search), however it does not run on customer content.

Azure Security Center monitoring: Not applicable

Responsibility: Not applicable

8.2: Pre-scan files to be uploaded to non-compute Azure resources

Guidance: Pre-scan any content being uploaded to non-compute Azure resources, such as Cognitive Search, Blob Storage, Azure SQL Database, and so on.

It is your responsibility to pre-scan any content being uploaded to non-compute Azure resources. Microsoft cannot access customer data, and therefore cannot conduct anti-malware scans of customer content on your behalf.

Azure Security Center monitoring: Not applicable

Responsibility: Customer

8.3: Ensure antimalware software and signatures are updated

Guidance: Not applicable to Cognitive Search. It does not allow for anti-malware solutions to be installed on its resources. For the underlying platform Microsoft handles updating any anti-malware software and signatures.

For any compute resources that are owned by your organization and used in your search solution, follow recommendations in Security Center, Compute & Apps to ensure all endpoints are up to date with the latest signatures. For Linux, use a third-party anti-malware solution.

Azure Security Center monitoring: Not applicable

Responsibility: Shared

Data recovery

For more information, see the [Azure Security Benchmark: Data recovery](#).

9.1: Ensure regular automated back ups

Guidance: Content stored in a search service cannot be backed up through Azure Backup or any other built-in mechanism, but you can rebuild an index from application source code and primary data sources, or build a custom tool to retrieve and store indexed content.

- [GitHub Index-backup-restore sample](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.2: Perform complete system backups and backup any customer-managed keys

Guidance: Cognitive Search currently doesn't support automated backup for data in a search service and must be backed up via a manual process. You can also backup customer-managed keys in Azure Key Vault.

- [Back up and restore an Azure Cognitive Search index](#)
- [How to backup Key Vault keys in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.3: Validate all backups including customer-managed keys

Guidance: Cognitive Search currently doesn't support automated backup for data in a search service and must be backed up and restored via a manual process. Periodically perform data restoration of content you have manually backed up to ensure the end-to-end integrity of your backup process.

- [Back up and restore an Azure Cognitive Search index](#)

- [How to restore Key Vault keys in Azure](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

9.4: Ensure protection of backups and customer-managed keys

Guidance: Cognitive Search currently does not support automated backup for data in a search service and must be backed up via a manual process. You can also backup customer-managed keys in Azure Key Vault.

Enable soft delete and purge protection in Key Vault to protect keys against accidental or malicious deletion. If Azure Storage is used to store manual backups, enable soft delete to save and recover your data when blobs or blob snapshots are deleted.

- [Back up and restore an Azure Cognitive Search index](#)
- [How to enable soft delete and purge protection in Key Vault](#)
- [Soft delete for Azure Blob storage](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

Incident response

For more information, see the [Azure Security Benchmark: Incident response](#).

10.1: Create an incident response guide

Guidance: Develop an incident response guide for your organization. Ensure there are written incident response plans that define all the roles of personnel as well as the phases of incident handling and management from detection to post-incident review.

- [Guidance on building your own security incident response process](#)
- [Microsoft Security Response Center's Anatomy of an Incident](#)
- [Customer may also leverage NIST's Computer Security Incident Handling Guide to aid in the creation of their own incident response plan](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.2: Create an incident scoring and prioritization procedure

Guidance: Security Center assigns a severity to each alert to help you prioritize which alerts should be investigated first. The severity is based on how confident Security Center is in the finding or the analytically used to issue the alert as well as the confidence level that there was malicious intent behind the activity that led to the alert.

Additionally, mark subscriptions using tags and create a naming system to identify and categorize Azure resources, especially those processing sensitive data. It's your responsibility to prioritize the remediation of alerts based on the criticality of the Azure resources and environment where the incident occurred.

- [Use tags to organize your Azure resources](#)
- [Security alerts in Azure Security Center](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

10.3: Test security response procedures

Guidance: Conduct exercises to test your systems' incident response capabilities on a regular cadence. Identify weak points and gaps and revise plan as needed.

- [Refer to NIST's publication, "Guide to Test, Training, and Exercise Programs for IT Plans and Capabilities"](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.4: Provide security incident contact details and configure alert notifications for security incidents

Guidance: Security incident contact information will be used by Microsoft to contact you if the Microsoft Security Response Center (MSRC) discovers that your data has been accessed by an unlawful or unauthorized party. Review incidents after the fact to ensure that issues are resolved.

- [How to set the Azure Security Center Security Contact](#)

Azure Security Center monitoring: Yes

Responsibility: Customer

10.5: Incorporate security alerts into your incident response system

Guidance: Export your Security Center alerts and recommendations using the Continuous Export feature. Continuous Export allows you to export alerts and recommendations either manually or on a continuous basis. You can use the Security Center data connector to stream the alerts to Azure Sentinel.

- [How to configure continuous export](#)
- [How to stream alerts into Azure Sentinel](#)

Azure Security Center monitoring: Not applicable

Responsibility: Customer

10.6: Automate the response to security alerts

Guidance: Use the Workflow Automation feature in Azure Security Center to automatically trigger responses via "Logic Apps" on security alerts and recommendations.

- [How to configure Workflow Automation and Logic Apps](#)

Azure Security Center monitoring: Currently not available

Responsibility: Customer

Penetration tests and red team exercises

For more information, see the [Azure Security Benchmark: Penetration tests and red team exercises](#).

11.1: Conduct regular penetration testing of your Azure resources and ensure remediation of all critical security findings

Guidance: Follow the Microsoft Cloud Penetration Testing Rules of Engagement to ensure your penetration tests are not in violation of Microsoft policies. Use Microsoft's strategy and execution of Red Teaming and live site penetration testing against Microsoft-managed cloud infrastructure, services, and applications.

- [Penetration Testing Rules of Engagement](#)
- [Microsoft Cloud Red Teaming](#)

Azure Security Center monitoring: Not applicable

Responsibility: Shared

Next steps

- See the [Azure security benchmark](#)
- Learn more about [Azure security baselines](#)

Create a basic search index in Azure Cognitive Search

10/4/2020 • 12 minutes to read • [Edit Online](#)

In Azure Cognitive Search, a *search index* stores searchable content used for full text and filtered queries. An index is defined by a schema and saved to the service, with data import following as a second step.

Indexes contain *documents*. Conceptually, a document is a single unit of searchable data in your index. A retailer might have a document for each product, a news organization might have a document for each article, 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 physical structure of an index is determined by the schema, with fields marked as "searchable" resulting in an inverted index created for that field.

You can create an index with the following tools and APIs:

- In the Azure portal, use [Add Index or Import data](#) wizard
- Using the [Create Index \(REST API\)](#)
- Using the [.NET SDK](#)

It's easier to learn with a portal tool. The portal enforces requirements and schema rules for specific data types, such as disallowing full text search capabilities on numeric fields. Once you have a workable index, you can transition to code by retrieving the JSON definition from the service using [Get Index \(REST API\)](#) and adding it to your solution.

Recommended workflow

Arriving at a final index design is an iterative process. It's common to start with the portal to create the initial index and then switch to code to place the index under source control.

1. Determine whether you can use [Import data](#). The wizard performs all-in-one indexer-based indexing if the source data is from a [supported data source type in Azure](#).
2. If you can't use [Import data](#), start with [Add Index](#) to define the schema.



3. Provide a name and key used to uniquely identify each search document in the index. The key is mandatory and must be of type Edm.String. During import, you should plan on mapping a unique field in source data to this field.

The portal gives you an `id` field for the key. To override the default `id`, create a new field (for example, a new field definition called `HotelID`) and then select it in **Key**.

Index name * ✓

Key * ✓

Suggester name Search mode ✓

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable	Analyzer	Suggester
<input checked="" type="checkbox"/> id	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Standard - Luc...	<input type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
HotelDescription_EN	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
HotelDescription_ES	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Spanish - Micr...	<input type="checkbox"/>

Create

- Add more fields. The portal shows you which [field attributes](#) are available for different data types. If you're new to index design, this is helpful.

If incoming data is hierarchical in nature, assign the [complex type](#) data type to represent the 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.

- Assign any [Analyzers](#) to string fields before the index is created. Do the same for [suggesters](#) if you want to enable autocomplete on specific fields.
- Click **Create** to build the physical structures in your search service.
- After an index is created, use additional commands to review definitions or add more elements.

Home > mydemo >
hotels-sample-index
index

Save Discard Refresh Create Search App (preview) Delete

Documents 50 Storage 343.67 kB

Search explorer Fields CORS Scoring profiles Index Definition (JSON)

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable	Analyzer	Suggester
<input checked="" type="checkbox"/> HotelId	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	English - Micro...	<input type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	French - Micro...	<input type="checkbox"/>
Category	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
Tags	Collection(Edm.String)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	English - Micro...	<input type="checkbox"/>
ParkingIncluded	Edm.Boolean	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			
LastRenovationDate	Edm.DateTime	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			

- Download the index schema using [Get Index \(REST API\)](#) and a web testing tool like [Postman](#). You now have a JSON representation of the index that you can adapt for code.
- [Load your index with data](#). Azure Cognitive Search accepts JSON documents. To load your data

programmatically, you can use Postman with JSON documents in the request payload. If your data is not easily expressed as JSON, this step will be the most labor intensive.

Once an index is loaded with data, most edits to existing fields will require that you drop and rebuild an index.

10. Query your index, examine results, and further iterate on the index schema until you begin to see the results you expect. You can use [Search explorer](#) or Postman to query your index.

During development, plan on frequent rebuilds. Because physical structures are created in the service, [dropping and recreating indexes](#) is necessary for most modifications to an existing field definition. You might consider working with a subset of your data to make rebuilds go faster.

TIP

Code, rather than a portal approach, is recommended for working on index design and data import simultaneously. As an alternative, tools like [Postman and the REST API](#) are helpful for proof-of-concept testing when development projects are still in early phases. You can make incremental changes to an index definition in a request body, and then send the request to your service to recreate an index using an updated schema.

Index schema

An index is required to have a name and one designated key field (of Edm.string) in the fields collection. The [*fields collection*](#) is typically the largest part of an index, where each field is named, typed, and attributed with allowable behaviors that determine how it is used.

Other elements include [suggesters](#), [scoring profiles](#), [analyzers](#) used to process strings into tokens according to linguistic rules or other characteristics supported by the analyzer, and [cross-origin remote scripting \(CORS\)](#) settings.

```
{
  "name": (optional on PUT; required on POST) "name_of_index",
  "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": [
    {
      "name": "name of suggester",
      "searchMode": "analyzingInfixMatching",
      "sourceFields": ["field1", "field2", ...]
    }
  ]
}
```

```

],
"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)
    },
    "freshness": {
      "boostingDuration": "..." (value representing timespan leading to now over which boosting occurs)
    },
    "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)
    },
    "tag": {
      "tagsParameter": "..." (parameter to be passed in queries to specify a list of tags to compare against target fields)
    }
  }
],
"functionAggregation": (optional, applies only when functions are specified)
  "sum (default) | average | minimum | maximum | firstMatching"
},
],
"analyzers":(optional)[ ... ],
"charFilters":(optional)[ ... ],
"tokenizers":(optional)[ ... ],
"tokenFilters":(optional)[ ... ],
"defaultScoringProfile": (optional) "...",
"corsOptions": (optional) {
  "allowedOrigins": ["*"] | ["origin_1", "origin_2", ...],
  "maxAgeInSeconds": (optional) max_age_in_seconds (non-negative integer)
},
"encryptionKey":(optional){
  "keyVaultUri": "azure_key_vault_uri",
  "keyVaultKeyName": "name_of.azure_key_vault_key",
  "keyVaultKeyVersion": "version_of.azure_key_vault_key",
  "accessCredentials":(optional){
    "applicationId": "azure_active_directory_application_id",
    "applicationSecret": "azure_active_directory_application_authentication_key"
  }
}
}

```

Fields collection and field attributes

Fields have a name, a type that classifies the stored data, and attributes that specify how the field is used.

Data types

TYPE	DESCRIPTION
Edm.String	Text that can optionally be tokenized for full-text search (word-breaking, stemming, and so forth).
Collection(Edm.String)	A list of strings that can optionally be tokenized for full-text search. There is no theoretical upper limit on the number of items in a collection, but the 16 MB upper limit on payload size applies to collections.
Edm.Boolean	Contains true/false values.
Edm.Int32	32-bit integer values.
Edm.Int64	64-bit integer values.
Edm.Double	Double-precision numeric data.
Edm.DateTimeOffset	Date time values represented in the OData V4 format (for example, <code>yyyy-MM-ddTHH:mm:ss.ffffZ</code> or <code>yyyy-MM-ddTHH:mm:ss.ffff[+/-]HH:mm</code>).
Edm.GeographyPoint	A point representing a geographic location on the globe.

For more information, see [supported data types](#).

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 "facetatable".

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 <code>Edm.String</code> or <code>Collection(Edm.String)</code> do not undergo word-breaking, so comparisons are for exact matches only. For example, if you set such a field f to "sunny day", <code>\$filter=f eq 'sunny'</code> will find no matches, but <code>\$filter=f eq 'sunny day'</code> will.
"sortable"	By default the system sorts results by score, but you can configure sort based on fields in the documents. Fields of type <code>Collection(Edm.String)</code> cannot be "sortable".

ATTRIBUTE	DESCRIPTION
"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 <code>Edm.GeographyPoint</code> . Fields of type <code>Edm.String</code> 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 <code>Edm.String</code> .
"retrievable"	Determines whether the field can be returned in a search result. This is useful when you want to use a field (such as <i>profit margin</i>) as a filter, sorting, or scoring mechanism, but do not want the field to be visible to the end user. This attribute must be <code>true</code> for <code>key</code> 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.

analyzers

The analyzers element sets the name of the language analyzer to use for the field. For more information about the range of analyzers available to you, see [Adding analyzers to an Azure Cognitive Search index](#). Analyzers can only be used with searchable fields. Once the analyzer is assigned to a field, it cannot be changed unless you rebuild the index.

suggesters

A suggester is a section of the schema that defines which fields in an index are used to support auto-complete or type-ahead queries in searches. Typically, partial search strings are sent to the [Suggestions \(REST API\)](#) while the user is typing a search query, and the API returns a set of suggested documents or phrases.

Fields added to a suggester are used to build type-ahead search terms. All of the search terms are created during indexing and stored separately. For more information about creating a suggester structure, see [Add suggesters](#).

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.

The following options 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 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.

scoringProfiles

A [scoring profile](#) is a section of the schema that defines custom scoring behaviors that let you influence which items appear higher in the search results. Scoring profiles are made up of field weights and functions. To use them, you specify a profile by name on the query string.

A default scoring profile operates behind the scenes to compute a search score for every item in a result set. You can use the internal, unnamed scoring profile. Alternatively, set **defaultScoringProfile** to use a custom profile as the default, invoked whenever a custom profile is not specified on the query string.

Attributes and index size (storage implications)

The size of an index is determined by the size of the documents you upload, plus index configuration, such as whether you include suggesters, and how you set attributes on individual fields.

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. 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 artificial, we can refer to them for broad comparisons of how attributes affect storage. Does setting " retrievable" increase index size? No. Does adding fields to a **suggester** increase index size? Yes.

Indexes that support filter and sort are proportionally larger than indexes supporting just full text search. This is because filter and sort operations scan for exact matches, requiring the presence of verbatim text strings. In contrast, searchable fields supporting full-text queries use inverted indexes, which are populated with tokenized terms that consume less space than whole documents.

NOTE

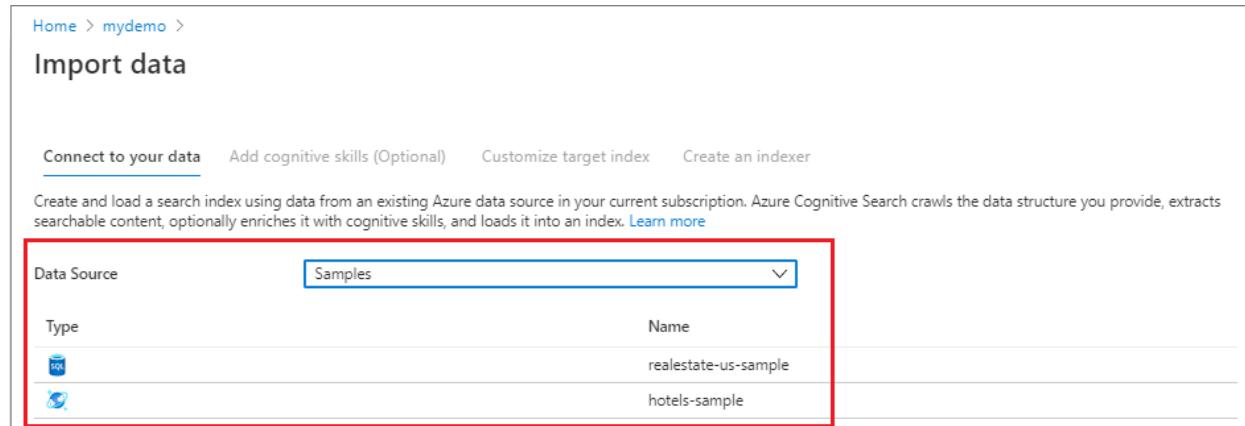
Storage architecture is considered an implementation detail of Azure Cognitive Search and could change without notice. There is no guarantee that current behavior will persist in the future.

Next steps

With an understanding of index composition, you can continue in the portal to create your first index. We recommend starting with **Import data** wizard, choosing either the *realestate-us-sample* or *hotels-sample* hosted data sources.

[Import data wizard \(portal\)](#)

For both data sets, the wizard can infer an index schema, import the data, and output a searchable index that you can query using Search Explorer. Find these data sources in the **Connect to your data** page of the **Import data** wizard.



Home > mydemo >

Import data

Connect to your data Add cognitive skills (Optional) Customize target index Create an indexer

Create and load a search index using data from an existing Azure data source in your current subscription. Azure Cognitive Search crawls the data structure you provide, extracts searchable content, optionally enriches it with cognitive skills, and loads it into an index. [Learn more](#)

Data Source	Samples
Type	realestate-us-sample
	hotels-sample

How to create an index for multiple languages in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

Indexes can include fields containing content from multiple languages, for example, creating individual fields for language-specific strings. For best results during indexing and querying, assign a language analyzer that provides the appropriate linguistic rules.

Azure Cognitive Search offers a large selection of language analyzers from both Lucene and Microsoft that can be assigned to individual fields using the Analyzer property. You can also specify a language analyzer in the portal, as described in this article.

Add analyzers to fields

A language analyzer is specified when a field is created. Adding an analyzer to an existing field definition requires overwriting (and reloading) the index, or creating a new field identical to the original, but with an analyzer assignment. You could then delete the unused field at your convenience.

1. Sign in to the [Azure portal](#) and find your search service.
2. Click **Add index** in the command bar at the top of the service dashboard to start a new index, or open an existing index to set an analyzer on new fields you're adding to an existing index.
3. Start a field definition by providing a name.
4. Choose the Edm.String data type. Only string fields are full-text searchable.
5. Set the **Searchable** attribute to enable the Analyzer property. A field must be text-based in order to make use of a language analyzer.
6. Choose one of the available analyzers.

Add index

* Index name [?](#)
test-index-2

* Key [?](#)
id

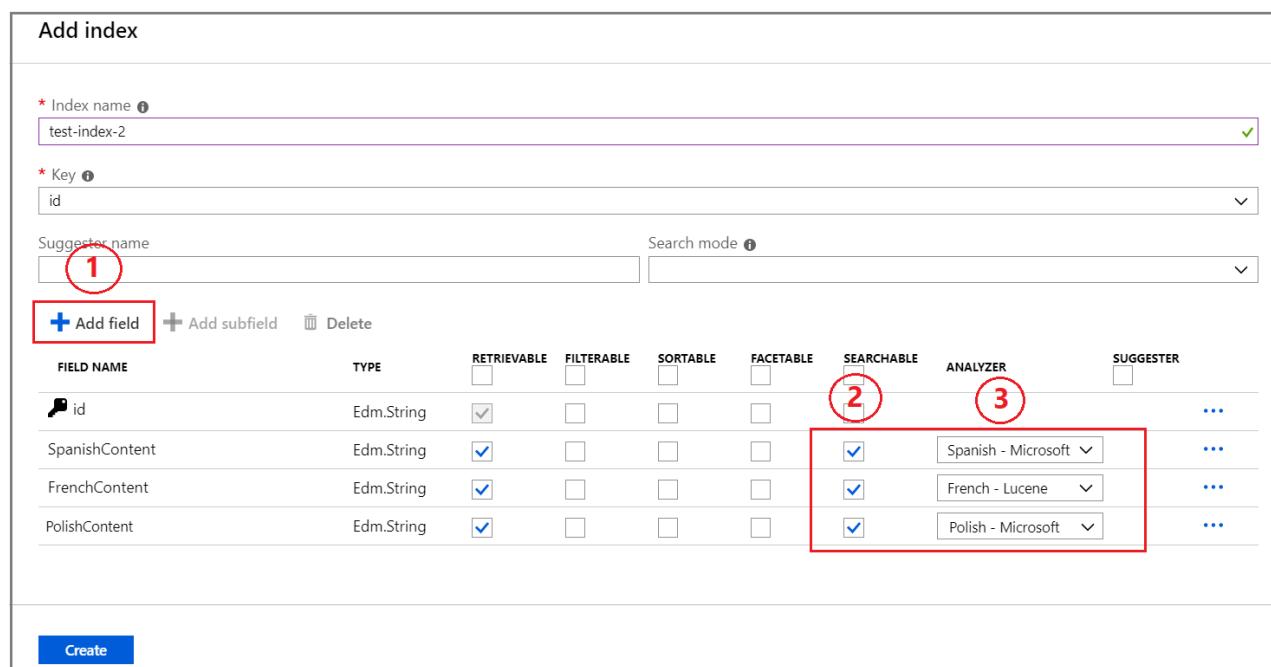
Suggester name [?](#) Search mode [?](#)

1

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACTETABLE	SEARCHABLE	ANALYZER	SUGGESTER
id	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Spanish - Microsoft	<input type="checkbox"/>
SpanishContent	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	French - Lucene	<input type="checkbox"/>
FrenchContent	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Polish - Microsoft	<input type="checkbox"/>
PolishContent	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>		<input type="checkbox"/>

2 **3**

Create



By default, all searchable fields use the [Standard Lucene analyzer](#) which is language-agnostic. To view the full list of supported analyzers, see [Add language analyzers to an Azure Cognitive Search index](#).

In the portal, analyzers are intended to be used as-is. If you require customization or a specific configuration of

filters and tokenizers, you should [create a custom analyzer](#) in code. The portal does not support selecting or configuring custom analyzers.

Query language-specific fields

Once the language analyzer is selected for a field, it will be used with each indexing and search request for that field. When a query is issued against multiple fields using different analyzers, the query will be processed independently by the assigned analyzers for each field.

If the language of the agent issuing a query is known, a search request can be scoped to a specific field using the **searchFields** query parameter. The following query will be issued only against the description in Polish:

```
https://[service name].search.windows.net/indexes/[index name]/docs?  
search=darmowy&searchfields=PolishContent&api-version=2020-06-30
```

You can query your index from the portal, using [Search explorer](#) to paste in a query similar to the one shown above.

Boost language-specific fields

Sometimes the language of the agent issuing a query is not known, in which case the query can be issued against all fields simultaneously. If needed, preference for results in a certain language can be defined using [scoring profiles](#). In the example below, matches found in the description in English will be scored higher relative to matches in Polish and French:

```
"scoringProfiles": [  
  {  
    "name": "englishFirst",  
    "text": {  
      "weights": { "description_en": 2 }  
    }  
  }  
]
```

```
https://[service name].search.windows.net/indexes/[index name]/docs?  
search=Microsoft&scoringProfile=englishFirst&api-version=2020-06-30
```

Next steps

If you're a .NET developer, note that you can configure language analyzers using the [Azure Cognitive Search .NET SDK](#) and the [Analyzer](#) property.

Analyzers for text processing in Azure Cognitive Search

10/4/2020 • 10 minutes to read • [Edit Online](#)

An *analyzer* is a component of the [full text search engine](#) responsible for processing text in query strings and indexed documents. 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 with 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 regular expression or fuzzy search, do not 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 get 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.

Default analyzer

In Azure Cognitive Search queries, 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. Additionally, 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 predefined 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.

CATEGORY	DESCRIPTION
Predefined analyzers	<p>Offered as a finished product intended to be used as-is. There are two types: specialized and language. What makes them "predefined" is that you reference them by name, with no configuration or customization.</p> <p>Specialized (language-agnostic) analyzers are used when text inputs require specialized processing or minimal processing. Non-language predefined analyzers include Asciifolding, Keyword, Pattern, Simple, Stop, Whitespace.</p> <p>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.</p>
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 predefined analyzers, such as **Pattern** or **Stop**, support a limited set of configuration options. To set these options, you effectively create a custom analyzer, consisting of the predefined analyzer and one of the alternative options documented in [Predefined Analyzer Reference](#). As with any custom configuration, provide your new configuration with a name, such as *myPatternAnalyzer* to distinguish it from the Lucene Pattern analyzer.

How to specify 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.

- When creating a field definition in the [index](#), set the **analyzer** property to one of the following: a [predefined analyzer](#) such as `keyword`, a [language analyzer](#) such as `en.microsoft`, or a custom analyzer (defined in the same index schema).

```

"fields": [
{
  "name": "Description",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "analyzer": "en.microsoft",
  "indexAnalyzer": null,
  "searchAnalyzer": null
},

```

If you are using a [language analyzer](#), you must use the **analyzer** property to specify it. The **searchAnalyzer** and **indexAnalyzer** properties do not support language analyzers.

- Alternatively, set **indexAnalyzer** and **searchAnalyzer** to vary the analyzer for each workload. These properties are set together and replace the **analyzer** property, which must be null. You might use different analyzers for data preparation and retrieval if one of those activities required a specific transformation not needed by the other.

```
"fields": [
{
  "name": "Description",
  "type": "Edm.String",
  "retrievable": true,
  "searchable": true,
  "analyzer": null,
  "indexAnalyzer": "keyword",
  "searchAnalyzer": "whitespace"
},
```

3. For custom analyzers only, create an entry in the [analyzers] section of the index, and then assign your custom analyzer to the field definition per either of the previous two steps. For more information, see [Create Index](#) and also [Add custom analyzers](#).

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 **analyzer** or **indexAnalyzer** to a field that has already been physically created is not 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 [rebuild the index entirely](#) (you cannot 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 via additional **indexAnalyzer** and **searchAnalyzer** field properties. If unspecified, the analyzer set with the **analyzer** property is used for both indexing and searching. If **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 are not 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 simply 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).

```
{
  "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 are not 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.

```
{
  "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 additional 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.

```
{
  "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 cannot have different analyzers for indexing and search.

```
{
  "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.

This example assigns Microsoft English and French analyzers to description fields. It's a snippet taken from a larger definition of the hotels index, creating using the Hotel class in the hotels.cs file of the [DotNetHowTo](#) sample.

Call [Analyzer](#), specifying the [AnalyzerName](#) type providing a text analyzer supported in Azure Cognitive Search.

```
public partial class Hotel
{
  . . .

  [IsSearchable]
  [Analyzer(AnalyzerName.AsString.EnMicrosoft)]
  [JsonProperty("description")]
  public string Description { get; set; }

  [IsSearchable]
  [Analyzer(AnalyzerName.AsString.FrLucene)]
  [JsonProperty("description_fr")]
  public string DescriptionFr { get; set; }

  . . .
}
```

Define a custom analyzer

When customization or configuration is required, you will need to 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. For more examples, see [CustomAnalyzerTests.cs](#).

```
{  
    var definition = new Index()  
    {  
        Name = "hotels",  
        Fields = FieldBuilder.BuildForType<Hotel>(),  
        Analyzers = new[]  
        {  
            new CustomAnalyzer()  
            {  
                Name = "url-analyze",  
                Tokenizer = TokenizerName.UaxUrlEmail,  
                TokenFilters = new[] { TokenFilterName.Lowercase }  
            }  
        },  
    };  
  
    serviceClient.Indexes.Create(definition);
```

Next steps

- Review our comprehensive explanation of [how full text search works in Azure Cognitive Search](#). This article uses examples to explain behaviors that might seem counter-intuitive on the surface.
- Try additional query syntax from the [Search Documents](#) example section or from [Simple query syntax](#) in Search explorer in the portal.
- Learn how to apply [language-specific lexical analyzers](#).
- [Configure custom analyzers](#) for either minimal processing or specialized processing on individual fields.

See also

[Search Documents REST API](#)

[Simple query syntax](#)

[Full Lucene query syntax](#)

[Handle search results](#)

Add language analyzers to string fields in an Azure Cognitive Search index

10/4/2020 • 4 minutes to read • [Edit Online](#)

A *language analyzer* is a specific type of [text analyzer](#) that performs lexical analysis using the linguistic rules of the target language. Every searchable 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](#) is language-agnostic, the concept of using spaces and special characters (hyphens and slashes) to separate strings tends is more applicable to Western languages than non-Western ones.

For example, in Chinese, Japanese, Korean (CJK), and other Asian languages, a space is not 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, Finnish, 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.

Indexing with Microsoft analyzers is on average two to three times slower than their Lucene equivalents, depending on the language. Search performance should not 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

Configuring analyzers

Language analyzers are used as-is. For each field in the index definition, you can set the **analyzer** property to an analyzer name that specifies the language and linguistics stack (Microsoft or Lucene). The same analyzer will be applied when indexing and searching for that field. For example, you can have separate fields for English, French, and Spanish hotel descriptions that exist side by side in the same index.

NOTE

It is not possible to use a different language analyzer at indexing time than at query time for a field. That capability is reserved for [custom analyzers](#). For this reason, if you try to set the **searchAnalyzer** or **indexAnalyzer** properties to the name of a language analyzer, the REST API will return an error response. You must use the **analyzer** property instead.

Use the **searchFields** query parameter to specify which language-specific field to search against in your queries. You can review query examples that include the analyzer property in [Search Documents](#).

For more information about index properties, see [Create Index \(Azure Cognitive Search REST API\)](#). For more information about analysis in Azure Cognitive Search, see [Analyzers in Azure Cognitive Search](#).

Language analyzer list

Below is the list of supported languages together 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

LANGUAGE	MICROSOFT ANALYZER NAME	LUCENE ANALYZER NAME
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
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	

LANGUAGE	MICROSOFT ANALYZER NAME	LUCENE ANALYZER NAME
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	
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 Index \(Azure Cognitive Search REST API\)](#)
- [AnalyzerName Class](#)

Add custom analyzers to string fields in an Azure Cognitive Search index

10/4/2020 • 25 minutes to read • [Edit Online](#)

A *custom analyzer* is a specific type of [text analyzer](#) that consists of a user-defined combination of existing tokenizer and optional filters. By combining tokenizers and filters in new ways, you can customize text processing in the search engine to achieve specific outcomes. For example, you could create a custom analyzer with a *char filter* to remove HTML markup before text inputs are tokenized.

You can define multiple custom analyzers to vary the combination of filters, but each field can only use one analyzer for indexing analysis and one for search analysis. For an illustration of what a customer analyzer looks like, see [Custom analyzer example](#).

Overview

The role of a [full-text search engine](#), in simple terms, is to process and store documents in a way that enables efficient querying and retrieval. At a high level, it all comes down to extracting important words from documents, putting them in an index, and then using the index to find documents that match words of a given query. The process of extracting words from documents and search queries is called *lexical analysis*. Components that perform lexical analysis are called *analyzers*.

In Azure Cognitive Search, you can choose from a set of predefined language-agnostic analyzers in the [Analyzers](#) table or language-specific analyzers listed in [Language analyzers \(Azure Cognitive Search service REST API\)](#). You also have an option to define your own custom analyzers.

A custom analyzer allows you to take control over the process of converting text into indexable and searchable tokens. It's a user-defined configuration consisting of a single predefined tokenizer, one or more token filters, and one or more char filters. The tokenizer is responsible for breaking text into tokens, and the token filters for modifying tokens emitted by the tokenizer. Char filters are applied for to prepare input text before it is processed by the tokenizer. For instance, char filter can replace certain characters or symbols.

Popular scenarios enabled by custom analyzers include:

- 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 are not 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.

This page provides a list of supported analyzers, tokenizers, token filters, and char filters. You can also find a description of changes to the index definition with a usage example. For more background about the underlying technology leveraged in the Azure Cognitive Search implementation, see [Analysis package summary \(Lucene\)](#). For examples of analyzer configurations, see [Add analyzers in Azure Cognitive Search](#).

Validation rules

Names of analyzers, tokenizers, token filters, and char filters have to be unique and cannot be the same as any of the predefined analyzers, tokenizers, token filters, or char filters. See the [Property Reference](#) for names already in use.

Create custom analyzers

You can define custom analyzers at index creation time. The syntax for specifying a custom analyzer is described in this section. You can also familiarize yourself with the syntax by reviewing sample definitions in [Add analyzers in Azure Cognitive Search](#).

An analyzer definition includes a name, a type, one or more char filters, a maximum of one tokenizer, and one or more token filters for post-tokenization processing. Char filters are applied before tokenization. Token filters and char filters are applied from left to right.

The `tokenizer_name` is the name of a tokenizer, `token_filter_name_1` and `token_filter_name_2` are the names of token filters, and `char_filter_name_1` and `char_filter_name_2` are the names of char filters (see the [Tokenizers](#), [Token filters](#) and [Char filters](#) tables for valid values).

The analyzer definition is a part of the larger index. See [Create Index API](#) for information about the rest of the index.

```

"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,
    ...
  }
]

```

NOTE

Custom analyzers that you create are not exposed in the Azure portal. The only way to add a custom analyzer is through code that makes calls to the API when defining an index.

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:

```
{  
  "name": "name_of_index",  
  "fields": [ ],  
  "suggesters": [ ],  
  "scoringProfiles": [ ],  
  "defaultScoringProfile": (optional) "...",  
  "corsOptions": (optional) { },  
  "analyzers":(optional)[ ],  
  "charFilters":(optional)[ ],  
  "tokenizers":(optional)[ ],  
  "tokenFilters":(optional)[ ]  
}
```

Definitions for char filters, tokenizers, and token filters are added to the index only if you are setting custom options. To use an existing filter or tokenizer as-is, specify it by name in the analyzer definition.

Test custom analyzers

You can use the [Test Analyzer operation](#) in the [REST API](#) to see how an analyzer breaks given text into tokens.

Request

```
POST https://[search service name].search.windows.net/indexes/[index name]/analyze?api-version=[api-version]  
Content-Type: application/json  
api-key: [admin key]  
  
{  

```

Response

```
{
  "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 char filter is defined, it cannot 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:

```
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.

Analyzer reference

The tables below list the configuration properties for the analyzers, tokenizers, token filters, and char filter section of an index definition. The structure of an analyzer, tokenizer, or filter in your index is composed of these attributes. For value assignment information, see the [Property Reference](#).

Analyzers

For analyzers, index attributes vary depending on the whether you're using predefined or custom analyzers.

Predefined Analyzers

TYPE	DESCRIPTION
Name	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	DESCRIPTION
Type	Analyzer type from the list of supported analyzers. See the analyzer_type column in the Analyzers table below.
Options	Must be valid options of a predefined analyzer listed in the Analyzers table below.

Custom Analyzers

TYPE	DESCRIPTION
Name	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	Set to either one of predefined char filters listed in the Char Filters table or a custom char filter specified in the index definition.
Tokenizer	Required. Set to either one of predefined tokenizers listed in the Tokenizers table below or a custom tokenizer specified in the index definition.
TokenFilters	Set to either one of predefined token filters listed in the Token filters table or a custom token filter specified in the index definition.

NOTE

It's required that you configure your custom analyzer to not produce tokens longer than 300 characters. Indexing fails for documents with such tokens. To trim them or ignore them, use the [TruncateTokenFilter](#) and the [LengthTokenFilter](#) respectively. Check [Token filters](#) for reference.

Char Filters

A char filter is used to prepare input text before it is processed by the tokenizer. For instance, they can replace certain characters or symbols. You can have multiple char filters in a custom analyzer. Char filters run in the order in which they are listed.

TYPE	DESCRIPTION
Name	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	Char filter type from the list of supported char filters. See the char_filter_type column in the Char Filters table below.
Options	Must be valid options of a given Char Filters type.

Tokenizers

A tokenizer divides continuous text into a sequence of tokens, such as breaking a sentence into words.

You can specify exactly one tokenizer per custom analyzer. 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.

A custom analyzer can use a predefined tokenizer with either default or customized options.

TYPE	DESCRIPTION
Name	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	Tokenizer name from the list of supported tokenizers. See tokenizer_type column in the Tokenizers table below.
Options	Must be valid options of a given tokenizer type listed in the Tokenizers table below.

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 are listed.

TYPE	DESCRIPTION
Name	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	Token filter name from the list of supported token filters. See token_filter_type column in the Token filters table below.
Options	Must be Token filters of a given token filter type.

Property reference

This section provides the valid values for attributes specified in the definition of a custom analyzer, tokenizer, char filter, or token filter in your index. Analyzers, tokenizers, and filters that are implemented using Apache Lucene have links to the Lucene API documentation.

Predefined Analyzers Reference

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.

ANALYZER_NAME	ANALYZER_TYPE	DESCRIPTION AND OPTIONS
pattern	PatternAnalyzer	<p>Flexibly separates text into terms via a regular expression pattern.</p> <p>Options</p> <p>lowercase (type: bool) - Determines whether terms are lowercased. The default is true.</p> <p>pattern (type: string) - A regular expression pattern to match token separators. The default is <code>\w+</code>, which matches non-word characters.</p> <p>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</p> <p>stopwords (type: string array) - A list of stopwords. The default is an empty list.</p>
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	<p>Standard Lucene analyzer, composed of the standard tokenizer, lowercase filter, and stop filter.</p> <p>Options</p> <p>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.</p> <p>stopwords (type: string array) - A list of stopwords. The default is an empty list.</p>
standardascifolding.lucene	(type applies only when options are available)	Standard analyzer with Ascii folding filter.
stop	StopAnalyzer	<p>Divides text at non-letters, applies the lowercase and stopword token filters.</p> <p>Options</p> <p>stopwords (type: string array) - A list of stopwords. The default is a predefined list for English.</p>
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 is no associated #Microsoft.Azure.Search type.

Char Filters Reference

In the table below, the character filters that are implemented using Apache Lucene are linked to the Lucene API documentation.

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.
mapping	MappingCharFilter	<p>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.</p> <p>Options</p> <p>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.</p>
pattern_replace	PatternReplaceCharFilter	<p>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".</p> <p>Options</p> <p>pattern (type: string) - Required.</p> <p>replacement (type: string) - Required.</p>

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 is no associated #Microsoft.Azure.Search type.

Tokenizers Reference

In the table below, the tokenizers that are implemented using Apache Lucene are linked to the Lucene API documentation.

TOKENIZER_NAME	TOKENIZER_TYPE ¹	DESCRIPTION AND OPTIONS
classic	ClassicTokenizer	<p>Grammar based tokenizer that is suitable for processing most European-language documents.</p> <p>Options</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.</p>
edgeNGram	EdgeNGramTokenizer	<p>Tokenizes the input from an edge into n-grams of given size(s).</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram.</p> <p>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.</p>
keyword_v2	KeywordTokenizerV2	<p>Emits the entire input as a single token.</p> <p>Options</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 256, maximum: 300. Tokens longer than the maximum length are split.</p>
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.

TOKENIZER_NAME	TOKENIZER_TYPE	DESCRIPTION AND OPTIONS
microsoft_language_tokenizer	MicrosoftLanguageTokenizer	<p>Divides text using language-specific rules.</p> <p>Options</p> <p>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.</p> <p>isSearchTokenizer (type: bool) - Set to true if used as the search tokenizer, set to false if used as the indexing tokenizer.</p> <p>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", "hindu", "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" </p>

TOKENIZER_NAME	TOKENIZER_TYPE	DESCRIPTION AND OPTIONS
microsoft_language_stemming_tokenizer	MicrosoftLanguageStemmingTokenizer	<p>Divides text using language-specific rules and reduces words to their base forms</p> <p>Options</p> <p>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.</p> <p>isSearchTokenizer (type: bool) - Set to true if used as the search tokenizer, set to false if used as the indexing tokenizer.</p> <p>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", "marathi", "norwegianBokmaal", "polish", "portuguese", "portugueseBrazilian", "punjabi", "romanian", "russian", "serbianCyrillic", "serbianLatin", "slovak", "slovenian", "spanish", "swedish", "tamil", "telugu", "turkish", "ukrainian", "urdu"</p>
nGram	NGramTokenizer	<p>Tokenizes the input into n-grams of the given size(s).</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum: 300. Must be greater than minGram.</p> <p>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.</p>

TOKENIZER_NAME	TOKENIZER_TYPE	DESCRIPTION AND OPTIONS
path_hierarchy_v2	PathHierarchyTokenizerV2	<p>Tokenizer for path-like hierarchies.</p> <p>Options</p> <p>delimiter (type: string) - Default: '/.</p> <p>replacement (type: string) - If set, replaces the delimiter character. Default same as the value of delimiter</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 300, maximum: 300. Paths longer than maxTokenLength are ignored.</p> <p>reverse (type: bool) - If true, generates token in reverse order. Default: false.</p> <p>skip (type: bool) - Initial tokens to skip. The default is 0.</p>
pattern	PatternTokenizer	<p>This tokenizer uses regex pattern matching to construct distinct tokens.</p> <p>Options</p> <p>pattern (type: string) - Regular expression pattern to match token separators. The default is <code>\w+</code>, which matches non-word characters.</p> <p>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</p> <p>group (type: int) - Which group to extract into tokens. The default is -1 (split).</p>
standard_v2	StandardTokenizerV2	<p>Breaks text following the Unicode Text Segmentation rules.</p> <p>Options</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.</p>
uax_url_email	UaxUrlEmailTokenizer	<p>Tokenizes urls and emails as one token.</p> <p>Options</p> <p>maxTokenLength (type: int) - The maximum token length. Default: 255, maximum: 300. Tokens longer than the maximum length are split.</p>

TOKENIZER_NAME	TOKENIZER_TYPE	DESCRIPTION AND OPTIONS
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 is no associated #Microsoft.Azure.Search type.

Token Filters Reference

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).
ascifolding	AsciiFoldingTokenFilter	<p>Converts alphabetic, numeric, and symbolic Unicode characters which are not in the first 127 ASCII characters (the "Basic Latin" Unicode block) into their ASCII equivalents, if one exists.</p> <p>Options</p> <p>preserveOriginal (type: bool) - If true, the original token is kept. The default is false.</p>
cjk_bigram	CjkBigramTokenFilter	<p>Forms bigrams of CJK terms that are generated from StandardTokenizer.</p> <p>Options</p> <p>ignoreScripts (type: string array) - Scripts to ignore. Allowed values include: "han", "hiragana", "katakana", "hangul". The default is an empty list.</p> <p>outputUnigrams (type: bool) - Set to true if you always want to output both unigrams and bigrams. The default is false.</p>
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.

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
classic	(type applies only when options are available)	Removes the English possessives, and dots from acronyms.
common_grams	CommonGramTokenFilter	<p>Construct bigrams for frequently occurring terms while indexing. Single terms are still indexed too, with bigrams overlaid.</p> <p>Options</p> <p>commonWords (type: string array) - The set of common words. The default is an empty list. Required.</p> <p>ignoreCase (type: bool) - If true, matching is case insensitive. The default is false.</p> <p>queryMode (type: bool) - Generates bigrams then removes common words and single terms followed by a common word. The default is false.</p>
dictionary_decompounder	DictionaryDecompounderTokenFilter	<p>Decomposes compound words found in many Germanic languages.</p> <p>Options</p> <p>wordList (type: string array) - The list of words to match against. The default is an empty list. Required.</p> <p>minWordSize (type: int) - Only words longer than this get processed. The default is 5.</p> <p>minSubwordSize (type: int) - Only subwords longer than this are outputted. The default is 2.</p> <p>maxSubwordSize (type: int) - Only subwords shorter than this are outputted. The default is 15.</p> <p>onlyLongestMatch (type: bool) - Add only the longest matching subword to output. The default is false.</p>

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
edgeNGram_v2	EdgeNGramTokenFilterV2	<p>Generates n-grams of the given size(s) from starting from the front or the back of an input token.</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram.</p> <p>side (type: string) - Specifies which side of the input the n-gram should be generated from. Allowed values: "front", "back"</p>
elision	ElisionTokenFilter	<p>Removes elisions. For example, "l'avion" (the plane) is converted to "avion" (plane).</p> <p>Options</p> <p>articles (type: string array) - A set of articles to remove. The default is an empty list. If there is no list of articles set, by default all French articles are removed.</p>
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.
keep	KeepTokenFilter	<p>A token filter that only keeps tokens with text contained in specified list of words.</p> <p>Options</p> <p>keepWords (type: string array) - A list of words to keep. The default is an empty list. Required.</p> <p>keepWordsCase (type: bool) - If true, lower case all words first. The default is false.</p>

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
keyword_marker	KeywordMarkerTokenFilter	<p>Marks terms as keywords.</p> <p>Options</p> <p>keywords (type: string array) - A list of words to mark as keywords. The default is an empty list. Required.</p> <p>ignoreCase (type: bool) - If true, lower case all words first. The default is false.</p>
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	<p>Removes words that are too long or too short.</p> <p>Options</p> <p>min (type: int) - The minimum number. Default: 0, maximum: 300.</p> <p>max (type: int) - The maximum number. Default: 300, maximum: 300.</p>
limit	Microsoft.Azure.Search.LimitTokenFilter	<p>Limits the number of tokens while indexing.</p> <p>Options</p> <p>maxTokenCount (type: int) - Max number of tokens to produce. The default is 1.</p> <p>consumeAllTokens (type: bool) - Whether all tokens from the input must be consumed even if maxTokenCount is reached. The default is false.</p>
lowercase	(type applies only when options are available)	Normalizes token text to lower case.
nGram_v2	NGramTokenFilterV2	<p>Generates n-grams of the given size(s).</p> <p>Options</p> <p>minGram (type: int) - Default: 1, maximum: 300.</p> <p>maxGram (type: int) - Default: 2, maximum 300. Must be greater than minGram.</p>

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
pattern_capture	PatternCaptureTokenFilter	<p>Uses Java regexes to emit multiple tokens, one for each capture group in one or more patterns.</p> <p>Options</p> <p>patterns (type: string array) - A list of patterns to match against each token. Required.</p> <p>preserveOriginal (type: bool) - Set to true to return the original token even if one of the patterns matches, default: true</p>
pattern_replace	PatternReplaceTokenFilter	<p>A token filter which applies a pattern to each token in the stream, replacing match occurrences with the specified replacement string.</p> <p>Options</p> <p>pattern (type: string) - Required.</p> <p>replacement (type: string) - Required.</p>
persian_normalization	(type applies only when options are available)	Applies normalization for Persian.
phonetic	PhoneticTokenFilter	<p>Create tokens for phonetic matches.</p> <p>Options</p> <p>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.</p> <p>See encoder for more information.</p> <p>replace (type: bool) - True if encoded tokens should replace original tokens, false if they should be added as synonyms. The default is true.</p>
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.

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
scandinavian_folding	(type applies only when options are available)	Folds Scandinavian characters åÄæÄÆ->a and öÖøØ->o. It also discriminates against use of double vowels aa, ae, ao, oe and oo, leaving just the first one.
shingle	ShingleTokenFilter	<p>Creates combinations of tokens as a single token.</p> <p>Options</p> <p>maxShingleSize (type: int) - Defaults to 2.</p> <p>minShingleSize (type: int) - Defaults to 2.</p> <p>outputUnigrams (type: bool) - if true, the output stream contains the input tokens (unigrams) as well as shingles. The default is true.</p> <p>outputUnigramsIfNoShingles (type: bool) - If true, override the behavior of outputUnigrams==false for those times when no shingles are available. The default is false.</p> <p>tokenSeparator (type: string) - The string to use when joining adjacent tokens to form a shingle. The default is " ".</p> <p>filterToken (type: string) - The string to insert for each position at which there is no token. The default is "_".</p>
snowball	SnowballTokenFilter	<p>Snowball Token Filter.</p> <p>Options</p> <p>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"</p>
sorani_normalization	SoraniNormalizationTokenFilter	<p>Normalizes the Unicode representation of Sorani text.</p> <p>Options</p> <p>None.</p>

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
stemmer	StemmerTokenFilter	<p>Language-specific stemming filter.</p> <p>Options</p> <p>language (type: string) - Allowed values include:</p> <ul style="list-style-type: none"> - "arabic" - "armenian" - "basque" - "brazilian" - "bulgarian" - "catalan" - "czech" - "danish" - "dutch" - "dutchKp" - "english" - "lightEnglish" - "minimalEnglish" - "possessiveEnglish" - "porter2" - "lovins" - "finnish" - "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	DESCRIPTION AND OPTIONS
stemmer_override	StemmerOverrideTokenFilter	<p>Any dictionary-Stemmed terms are marked as keywords, which prevents stemming down the chain. Must be placed before any stemming filters.</p> <p>Options</p> <p>rules (type: string array) - Stemming rules in the following format "word => stem" for example "ran => run". The default is an empty list. Required.</p>
stopwords	StopwordsTokenFilter	<p>Removes stop words from a token stream. By default, the filter uses a predefined stop word list for English.</p> <p>Options</p> <p>stopwords (type: string array) - A list of stopwords. Cannot be specified if a stopwordsList is specified.</p> <p>stopwordsList (type: string) - A predefined list of stopwords. Cannot 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". Cannot be specified if stopwords is specified.</p> <p>ignoreCase (type: bool) - If true, all words are lower cased first. The default is false.</p> <p>removeTrailing (type: bool) - If true, ignore the last search term if it's a stop word. The default is true.</p>

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
synonym	SynonymTokenFilter	<p>Matches single or multi word synonyms in a token stream.</p> <p>Options</p> <p>synonyms (type: string array) - Required. List of synonyms in one of the following two formats:</p> <ul style="list-style-type: none"> -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. <p>ignoreCase (type: bool) - Case-folds input for matching. The default is false.</p> <p>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</p> <p>- If false, the following list: incredible, unbelievable, fabulous, amazing are equivalent to: incredible, unbelievable, fabulous, amazing => incredible.</p>
trim	(type applies only when options are available)	Trims leading and trailing whitespace from tokens.
truncate	TruncateTokenFilter	<p>Truncates the terms into a specific length.</p> <p>Options</p> <p>length (type: int) - Default: 300, maximum: 300. Required.</p>
unique	UniqueTokenFilter	<p>Filters out tokens with same text as the previous token.</p> <p>Options</p> <p>onlyOnSamePosition (type: bool) - If set, remove duplicates only at the same position. The default is true.</p>
uppercase	(type applies only when options are available)	Normalizes token text to upper case.

TOKEN_FILTER_NAME	TOKEN_FILTER_TYPE	DESCRIPTION AND OPTIONS
word_delimiter	WordDelimiterTokenFilter	<p>Splits words into subwords and performs optional transformations on subword groups.</p> <p>Options</p> <p>generateWordParts (type: bool) - Causes parts of words to be generated, for example "AzureSearch" becomes "Azure" "Search". The default is true.</p> <p>generateNumberParts (type: bool) - Causes number subwords to be generated. The default is true.</p> <p>catenateWords (type: bool) - Causes maximum runs of word parts to be catenated, for example "Azure-Search" becomes "AzureSearch". The default is false.</p> <p>catenateNumbers (type: bool) - Causes maximum runs of number parts to be catenated, for example "1-2" becomes "12". The default is false.</p> <p>catenateAll (type: bool) - Causes all subword parts to be catenated, e.g "Azure-Search-1" becomes "AzureSearch1". The default is false.</p> <p>splitOnCaseChange (type: bool) - If true, splits words on caseChange, for example "AzureSearch" becomes "Azure" "Search". The default is true.</p> <p>preserveOriginal - Causes original words to be preserved and added to the subword list. The default is false.</p> <p>splitOnNumerics (type: bool) - If true, splits on numbers, for example "Azure1Search" becomes "Azure" "1" "Search". The default is true.</p> <p>stemEnglishPossessive (type: bool) - Causes trailing "s" to be removed for each subword. The default is true.</p> <p>protectedWords (type: string array) - Tokens to protect from being delimited. The default is an empty list.</p>

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 \(Azure Cognitive Search REST API\)](#)

Synonyms in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

Synonyms in search engines associate equivalent terms that implicitly expand the scope of a query, without the user having to actually provide the term. For example, given the term "dog" and synonym associations of "canine" and "puppy", any documents containing "dog", "canine" or "puppy" will fall within the scope of the query.

In Azure Cognitive Search, synonym expansion is done at query time. You can add synonym maps to a service with no disruption to existing operations. You can add a **synonymMaps** property to a field definition without having to rebuild the index.

Create synonyms

There is no portal support for creating synonyms but you can use the REST API or .NET SDK. To get started with REST, we recommend [using Postman](#) and formulation of requests using this API: [Create Synonym Maps](#). For C# developers, you can get started with [Add Synonyms in Azure Cognitive Searching using C#](#).

Optionally, if you are using [customer-managed keys](#) for service-side encryption-at-rest, you can apply that protection to the contents of your synonym map.

Use synonyms

In Azure Cognitive Search, synonym support is based on synonym maps that you define and upload to your service. These maps constitute an independent resource (like indexes or data sources), and can be used by any searchable field in any index in your search service.

Synonym maps and indexes are maintained independently. Once you define a synonym map and upload it to your service, you can enable the synonym feature on a field by adding a new property called **synonymMaps** in the field definition. Creating, updating, and deleting a synonym map is always a whole-document operation, meaning that you cannot create, update or delete parts of the synonym map incrementally. Updating even a single entry requires a reload.

Incorporating synonyms into your search application is a two-step process:

1. Add a synonym map to your search service through the APIs below.
2. Configure a searchable field to use the synonym map in the index definition.

You can create multiple synonym maps for your search application (for example, by language if your application supports a multi-lingual customer base). Currently, a field can only use one of them. You can update a field's **synonymMaps** property at any time.

SynonymMaps Resource APIs

[Add or update a synonym map under your service, using POST or PUT.](#)

Synonym maps are uploaded to the service via POST or PUT. 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 all other SKUs. Each rule can have up to 20 expansions.

Synonym maps must be in the Apache Solr format which is explained below. If you have an existing synonym dictionary in a different format and want to use it directly, please let us know on [UserVoice](#).

You can create a new synonym map using HTTP POST, as in the following example:

```
POST https://[servicename].search.windows.net/synonymmaps?api-version=2020-06-30
api-key: [admin key]

{
  "name": "mysynonymmap",
  "format": "solr",
  "synonyms": "
    USA, United States, United States of America\n
    Washington, Wash., WA => WA\n"
}
```

Alternatively, you can use PUT and specify the synonym map name on the URI. If the synonym map does not exist, it will be created.

```
PUT https://[servicename].search.windows.net/synonymmaps/mysynonymmap?api-version=2020-06-30
api-key: [admin key]

{
  "format": "solr",
  "synonyms": "
    USA, United States, United States of America\n
    Washington, Wash., WA => WA\n"
}
```

Apache Solr synonym format

The Solr format supports equivalent and explicit synonym mappings. Mapping rules adhere to the open-source synonym filter specification of Apache Solr, described in this document: [SynonymFilter](#). Below is a sample rule for equivalent synonyms.

```
USA, United States, United States of America
```

With the rule above, a search query "USA" will expand to "USA" OR "United States" OR "United States of America".

Explicit mapping is 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. Given the rule below, search queries "Washington", "Wash." or "WA" will all be rewritten to "WA". Explicit mapping only applies in the direction specified and does not rewrite the query "WA" to "Washington" in this case.

```
Washington, Wash., WA => WA
```

If you need to define synonyms that contain commas, you can escape them with a backslash, like in this example:

```
WA\\, USA, WA, Washington
```

Since backslash is itself a special character in other languages like JSON and C#, you will probably need to double-escape it. For example, the JSON sent to the REST API for the above synonym map would look like this:

```
{
  "format": "solr",
  "synonyms": "WA\\\\, USA, WA, Washington"
}
```

List synonym maps under your service.

```
GET https://[servicename].search.windows.net/synonymmaps?api-version=2020-06-30  
api-key: [admin key]
```

Get a synonym map under your service.

```
GET https://[servicename].search.windows.net/synonymmaps/mysynonymmap?api-version=2020-06-30  
api-key: [admin key]
```

Delete a synonyms map under your service.

```
DELETE https://[servicename].search.windows.net/synonymmaps/mysynonymmap?api-version=2020-06-30  
api-key: [admin key]
```

Configure a searchable field to use the synonym map in the index definition.

A new field property **synonymMaps** can be used to specify a synonym map to use for a searchable field. Synonym maps are service level resources and can be referenced by any field of an index under the service.

```
POST https://[servicename].search.windows.net/indexes?api-version=2020-06-30  
api-key: [admin key]
```

```
{
  "name": "myindex",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true
    },
    {
      "name": "name",
      "type": "Edm.String",
      "searchable": true,
      "analyzer": "en.lucene",
      "synonymMaps": [
        "mysynonymmap"
      ]
    },
    {
      "name": "name_jp",
      "type": "Edm.String",
      "searchable": true,
      "analyzer": "ja.microsoft",
      "synonymMaps": [
        "japanesesynonymmap"
      ]
    }
  ]
}
```

synonymMaps can be specified for searchable fields of the type 'Edm.String' or 'Collection(Edm.String)'.

NOTE

You can only have one synonym map per field. If you want to use multiple synonym maps, please let us know on [UserVoice](#).

Impact of synonyms on other search features

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.

Synonym feature applies to search queries and does not apply to filters or facets. Similarly, suggestions are based only on the original term; synonym matches do not appear in the response.

Synonym expansions do not 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](#)

Example: Add synonyms for Azure Cognitive Search in C#

10/4/2020 • 5 minutes to read • [Edit Online](#)

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. You learn how to:

- Create a synonym map using the [SynonymMap](#) class.
- Set the [SynonymMaps](#) 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. If Azure portal support for synonyms would be useful to you, please provide your feedback on the [UserVoice](#)

Prerequisites

Tutorial requirements include the following:

- [Visual Studio](#)
- [Azure Cognitive Search service](#)
- [Microsoft.Azure.Search .NET library](#)
- [How to use Azure Cognitive Search from a .NET Application](#)

Overview

Before-and-after queries demonstrate the value of synonyms. In this example, use a sample application that executes queries and returns results on a sample index. The sample application creates a small index named "hotels" populated with two documents. The application executes search queries using terms and phrases that do not appear in the index, enables the synonyms feature, then issues the same searches again. The code below demonstrates the overall flow.

```

static void Main(string[] args)
{
    SearchServiceClient serviceClient = CreateSearchServiceClient();

    Console.WriteLine("{0}", "Cleaning up resources...\n");
    CleanupResources(serviceClient);

    Console.WriteLine("{0}", "Creating index...\n");
    CreateHotelsIndex(serviceClient);

    ISearchIndexClient indexClient = serviceClient.Indexes.GetClient("hotels");

    Console.WriteLine("{0}", "Uploading documents...\n");
    UploadDocuments(indexClient);

    ISearchIndexClient indexClientForQueries = CreateSearchIndexClient();

    RunQueriesWithNonExistentTermsInIndex(indexClientForQueries);

    Console.WriteLine("{0}", "Adding synonyms...\n");
    UploadSynonyms(serviceClient);
    EnableSynonymsInHotelsIndex(serviceClient);
    Thread.Sleep(10000); // Wait for the changes to propagate

    RunQueriesWithNonExistentTermsInIndex(indexClientForQueries);

    Console.WriteLine("{0}", "Complete. Press any key to end application...\n");

    Console.ReadKey();
}

```

The steps to create and populate the sample index are explained in [How to use Azure Cognitive Search from a .NET Application](#).

"Before" queries

In `RunQueriesWithNonExistentTermsInIndex`, issue search queries with "five star", "internet", and "economy AND hotel".

```

Console.WriteLine("Search the entire index for the phrase \"five star\":\n");
results = indexClient.Documents.Search<Hotel>("\\"five star\\\"", parameters);
WriteDocuments(results);

Console.WriteLine("Search the entire index for the term 'internet':\n");
results = indexClient.Documents.Search<Hotel>("internet", parameters);
WriteDocuments(results);

Console.WriteLine("Search the entire index for the terms 'economy' AND 'hotel':\n");
results = indexClient.Documents.Search<Hotel>("economy AND hotel", parameters);
WriteDocuments(results);

```

Neither of the two indexed documents contain the terms, so we get the following output from the first `RunQueriesWithNonExistentTermsInIndex`.

```
Search the entire index for the phrase "five star":
```

```
no document matched
```

```
Search the entire index for the term 'internet':
```

```
no document matched
```

```
Search the entire index for the terms 'economy' AND 'hotel':
```

```
no document matched
```

Enable synonyms

Enabling synonyms is a two-step process. We first define and upload synonym rules and then 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.

```
var synonymMap = new SynonymMap()
{
    Name = "desc-synonymmap",
    Format = "solr",
    Synonyms = "hotel, motel\n
                internet,wifi\n
                five star=>luxury\n
                economy,inexpensive=>budget"
};

serviceClient.SynonymMaps.CreateOrUpdate(synonymMap);
```

A synonym map must conform to the open source standard [solr](#) format. The format is explained in [Synonyms in Azure Cognitive Search](#) under the section [Apache Solr synonym format](#).

2. Configure searchable fields to use the synonym map in the index definition. In [EnableSynonymsInHotelsIndex](#), we enable synonyms on two fields `category` and `tags` by setting the `synonymMaps` property to the name of the newly uploaded synonym map.

```
Index index = serviceClient.Indexes.Get("hotels");
index.Fields.First(f => f.Name == "category").SynonymMaps = new[] { "desc-synonymmap" };
index.Fields.First(f => f.Name == "tags").SynonymMaps = new[] { "desc-synonymmap" };

serviceClient.Indexes.CreateOrUpdate(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 `synonymMaps` property to an empty list.

```
index.Fields.First(f => f.Name == "category").SynonymMaps = new List<string>();
```

"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:

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.

Sample application source code

You can find the full source code of the sample application used in this walk through on [GitHub](#).

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](#)

How to model complex data types in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

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.

```
{
  "HotelId": "1",
  "HotelName": "Secret Point Motel",
  "Description": "Ideally located on the main commercial artery of the city in the heart of New York.",
  "Address": {
    "StreetAddress": "677 5th Ave",
    "City": "New York",
    "StateProvince": "NY"
  },
  "Rooms": [
    {
      "Description": "Budget Room, 1 Queen Bed (Cityside)",
      "Type": "Budget Room",
      "BaseRate": 96.99
    },
    {
      "Description": "Deluxe Room, 2 Double Beds (City View)",
      "Type": "Deluxe Room",
      "BaseRate": 150.99
    }
  ]
}
```

Creating 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.

The following example shows a JSON index schema with simple fields, collections, and complex types. Notice that within a complex type, each sub-field has a type and may have attributes, just as top-level fields do. The schema corresponds to the example data above. `Address` is a complex field that isn't a collection (a hotel has one address). `Rooms` is a complex collection field (a hotel has many rooms).

```
{
  "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": "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": "Type", "type": "Edm.String", "searchable": true },
        { "name": "BaseRate", "type": "Edm.Double", "filterable": true, "facetable": true }
      ]
    }
  ]
}
```

Updating 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 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.

Searching 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](#).

Selecting 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 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.

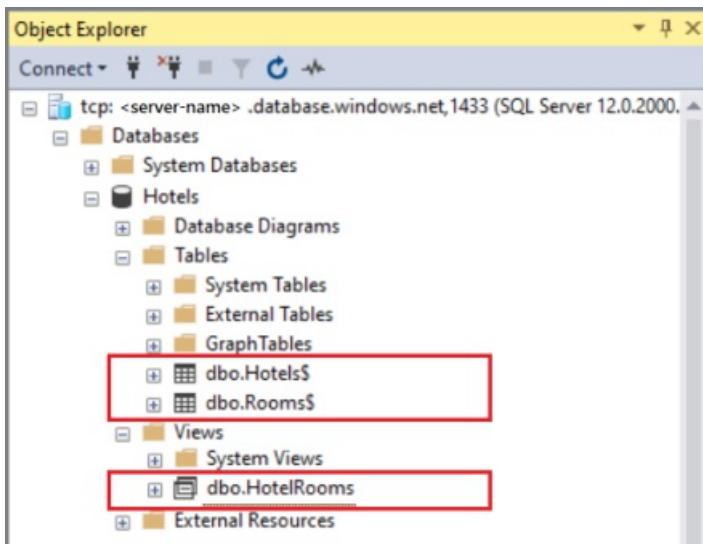
[Quickstart: portal wizard for import, indexing, and queries](#)

How to model relational SQL data for import and indexing in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

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'll 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 is a one-to-many relationship between the tables. In our approach, a view will provide 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 will return denormalized data, which doesn't work well in an Azure Cognitive Search scenario. Consider the following example that joins hotels and rooms.

```
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.

Fields from Hotels\$				Fields from Rooms\$			
HotelID	HotelName	Description	State	HotelID	Description	Type	BaseRate
181	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Standard Room, 1 Queen Bed (City View)	Standard Room	121.99
182	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Budget Room, 1 King Bed (Waterfront View)	Budget Room	88.99
183	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Standard Room, 2 Double Beds (Citside)	Standard Room	127.99
184	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Budget Room, 2 Double Beds (Citside)	Budget Room	96.99
185	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Budget Room, 1 Queen Bed (Mountain View)	Budget Room	63.99
186	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Standard Room, 1 Queen Bed (Mountain View)	Standard Room	124.99
187	2	Twin Dome Motel	The hotel is situated in a nineteenth century plaza...	FL	Standard Room, 1 Queen Bed (Citside)	Standard Room	117.99

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 will create 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. Fortunately, you can do this easily by adopting the techniques in this article.

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 have 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.

```
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.

```

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.

The screenshot shows the 'Views' node expanded, with 'dbo.HotelRooms' selected. Under 'Columns', the following fields are listed:

- 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)
- Rooms (nvarchar(max), null)**

- 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.

The screenshot shows the results of the query. The 'Results' tab is selected. The table has columns corresponding to the hotel fields. The 'Rooms' column contains JSON arrays for each hotel, which are highlighted with a red border. The status bar at the bottom right indicates '50 rows'.

HotelID	HotelName	Des...	De...	Ca...	T...	P...	S...	L...	R...	St...	C...	St...	Z...	Geo...	Rooms
1	Secret Point Motel	Th...	L...	B...	p...	0	1	1...	3...	6...	N.	NY	1...	[7...	[{"HotelID": "1", "Description": "Budget Room, 1 Queen Bed (Ctryside)", "De...}
2	Countryside Hotel	Sa...	\é...	B...	2...	0	1	1...	2...	6...	D.	NC	2...	[7...	[{"HotelID": "10", "Description": "Suite, 1 King Bed (Amenities)", "Descri...}
3	Regal Orb Resort & Spa	Yo...	V...	E...	fr...	1	0	1...	2...	2...	B.	W...	9...	[1...	[{"HotelID": "11", "Description": "Deluxe Room, 1 Queen Bed (Waterfront Vie...}
4	Winter Panorama Resort	Ne...	R...	I...	I...	0	0	1...	4...	9...	W	OR	9...	[1...	[{"HotelID": "12", "Description": "Deluxe Room, 1 King Bed (Ctryside)", "Descri...}
5	Historic Lion Resort	Un...	U...	B...	v...	0	1	1...	4...	3...	S.	M...	6...	[9...	[{"HotelID": "13", "Description": "Standard Room, 1 King Bed (Mountain Vie...}
6	Twin Vertex Hotel	Ne...	N...	E...	b...	0	0	1...	4...	1...	D.	TX	7...	[9...	[{"HotelID": "14", "Description": "Budget Room, 1 King Bed (Ctryside)", "Descri...}
7	Peaceful Market Hotel & Spa	Bo...	R...	R...	c...	1	0	2...	3...	1...	N.	NY	1...	[7...	[{"HotelID": "15", "Description": "Standard Room, 1 King Bed (Waterfront Vie...}
8	Double Sanctuary Resort	5"	5...	R...	v...	0	0	1...	4...	2...	S.	W...	9...	[1...	[{"HotelID": "16", "Description": "Suite, 2 Queen Beds (Amenities)", "Descri...}
9	Antiquity Hotel	Ele...	\é...	B...	r...	0	0	1...	4...	8...	N.	NY	1...	[7...	[{"HotelID": "17", "Description": "Budget Room, 2 Queen Beds (Waterfront Vi...}
10	Oceanside Resort	Ne...	N...	B...	v...	1	1	1...	4...	5...	T.	FL	3...	[8...	[{"HotelID": "18", "Description": "Standard Room, 1 Queen Bed (Ctryside)", "...}
11	Universe Motel	Bo...	R...	S...	r...	0	0	1...	2...	1...	R.	W...	9...	[1...	[{"HotelID": "19", "Description": "Deluxe Room, 1 Queen Bed (Waterfront Vie...}
12	Twin Dome Motel	Th...	L...	B...	p...	0	1	1...	3...	1...	S.	FL	3...	[8...	[{"HotelID": "2", "Description": "Suite, 2 Double Beds (Mountain View)", "De...}

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 is composed of a fixed set of items, as opposed to the multiple, arbitrary number of items allowed in a collection.

```
{  
  "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": false },  
    { "name": "ParkingIncluded", "type": "Edm.Boolean", "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": true  
},  
        { "name": "SleepsCount", "type": "Edm.Int32", "filterable": true, "facetable": true },  
        { "name": "SmokingAllowed", "type": "Edm.Boolean", "filterable": true, "facetable": true },  
        { "name": "Tags", "type": "Edm.Collection", "searchable": true }  
      ]  
    }  
  ]  
}
```

Given the previous result set and the above index schema, you have 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.

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.

Dashboard > Import data

Import data

* Connect to your data Add cognitive search (Optional) * Customize target index

* Index name ✓

* Key ▾

+ Add field + Add subfield Delete

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	Facetable	SEARCHABLE
HotelID	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
ZipCode	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
GeoCoordinates	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Rooms	Collection<...>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
HotelID	Edm.ComplexType	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Collection(Edm.ComplexType)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
	Collection(Edm.ComplexType)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Try the following quickstart to learn the basic steps of the Import data wizard.

[Quickstart: Create a search index using Azure portal](#)

Data import overview - Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

In Azure Cognitive Search, queries execute over your content loaded into and saved in a [search index](#). This article examines the two basic approaches for populating an index: *push* your data into the index programmatically, or point an [Azure Cognitive Search indexer](#) at a supported data source to *pull* in the data.

With either approach, the objective is to load data from an external data source into an Azure Cognitive Search index. Azure Cognitive Search will let you create an empty index, but until you push or pull data into it, it's not queryable.

NOTE

If [AI enrichment](#) is a solution requirement, you must use the pull model (indexers) to load an index. External processing is supported only through skillsets attached to an indexer.

Pushing data to an index

The push model, used to programmatically send your data to Azure Cognitive Search, is the most flexible approach. First, it has no restrictions on data source type. Any dataset composed of JSON documents can be pushed to an Azure Cognitive Search index, assuming each document in the dataset has fields mapping to fields defined in your index schema. Second, it has no restrictions on frequency of execution. You can push changes to an index as often as you like. For applications having very 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.

This approach is more flexible than the pull model because you can upload documents individually or in batches (up to 1000 per batch or 16 MB, whichever limit comes first). The push model also allows you to upload documents to Azure Cognitive Search regardless of where your data is.

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 \(REST API\)](#)
- [indexAction class](#) or [indexBatch class](#)

There is currently no tool support for pushing data via the portal.

For an introduction to each methodology, see [Quickstart: Create an Azure Cognitive Search index using PowerShell](#) or [C# Quickstart: Create an Azure Cognitive Search index using .NET SDK](#).

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.

In the REST API, issue HTTP POST requests with JSON request bodies to your Azure Cognitive Search index's endpoint URL. Each JSON object in the "value" array contains the document's key and specifies whether an indexing action adds, updates, or deletes document content. For a code example, see [Load documents](#).

In the .NET SDK, package up your data into an `IndexBatch` object. An `IndexBatch` encapsulates a collection of `IndexAction` objects, each of which contains a document and a property that tells Azure Cognitive Search what action to perform on that document. For a code example, see the [C# Quickstart](#).

@SEARCH.ACTION	DESCRIPTION	NECESSARY FIELDS FOR EACH DOCUMENT	NOTES
<code>upload</code>	An <code>upload</code> action is similar to an "upsert" where the document will be inserted if it is new and updated/replaced if it exists.	key, plus any other fields you wish to define	When updating/replacing an existing document, any field that is not specified in the request will have its field set to <code>null</code> . This occurs even when the field was previously set to a non-null value.
<code>merge</code>	Updates an existing document with the specified fields. If the document does not exist in the index, the merge will fail.	key, plus any other fields you wish to define	Any field you specify in a merge will replace the existing field in the document. In the .NET SDK, this includes fields of type <code>DataType.Collection(DataType.String)</code> . In the REST API, this includes fields of type <code>Collection(Edm.String)</code> . For example, if the document contains a field <code>tags</code> with value <code>["budget"]</code> and you execute a merge with value <code>["economy", "pool"]</code> for <code>tags</code> , the final value of the <code>tags</code> field will be <code>["economy", "pool"]</code> . It will not be <code>["budget", "economy", "pool"]</code> .
<code>mergeOrUpload</code>	This action behaves like <code>merge</code> if a document with the given key already exists in the index. If the document does not exist, it behaves like <code>upload</code> with a new document.	key, plus any other fields you wish to define	-
<code>delete</code>	Removes the specified document from the index.	key only	Any fields you specify other than the key field will be ignored. If you want to remove an individual field from a document, use <code>merge</code> instead and simply set the field explicitly to null.

Formulate your query

There are two ways to [search your index using the REST API](#). One way is to issue an HTTP POST request where your query parameters are defined in a JSON object in the request body. The other way is to issue an HTTP GET request where your query parameters are defined within the request URL. POST has more [relaxed limits](#) on the size of query parameters than GET. For this reason, we recommend using POST unless you have special circumstances where using GET would be more convenient.

For both POST and GET, you need to provide your *service name*, *index name*, and an *API version* in the request URL.

For GET, the *query string* at the end of the URL is where you provide the query parameters. See below for the URL format:

```
https://[service name].search.windows.net/indexes/[index name]/docs?[query string]&api-version=2019-05-06
```

The format for POST is the same, but with `api-version` in the query string parameters.

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:

- [Blob storage](#)
- [Table storage](#)
- [Azure Cosmos DB](#)
- [Azure SQL Database, SQL Managed Instance, and SQL Server on Azure VMs](#)

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 scheduling 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 for you 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 you want to make impact the schema directly, 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.

TIP

Numerous [Azure Cognitive Search code samples](#) include embedded or readily available datasets, offering an easy way to get started. The portal also provides a sample indexer and data source consisting of a small real estate dataset (named "realestate-us-sample"). When you run the preconfigured indexer on the sample data source, an index is created and loaded with documents that can then be queried in Search explorer or by code that you write.

See also

- [Indexer overview](#)
- [Portal walkthrough: create, load, query an index](#)

Import data wizard for Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

The Azure portal provides an **Import data** wizard on the Azure Cognitive Search dashboard for prototyping and loading an index. This article covers advantages and limitations of using the wizard, inputs and outputs, and some usage information. For hands-on guidance in stepping through the wizard using built-in sample data, see the [Create an Azure Cognitive Search index using the Azure portal](#) quickstart.

Operations that this wizard performs include:

- 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.
- 4 - Run the wizard to create objects, import data, set a schedule and other configuration options.

The wizard outputs a number of objects that are saved to your search service, which you can access programmatically or in other tools.

Advantages and limitations

Before you write 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.

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 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 providing data as JSON documents, are handled by the wizard.

Known limitations are summarized as follows:

- The wizard does not 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, you have to use the REST APIs or .NET SDK to retrieve and modify the structures.
- Source content must reside in a supported Azure 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 few built-in skills.
- A [knowledge store](#), which can be created by the wizard, is limited to a few default projections. If you want to save enriched documents created by the wizard, the blob container and tables come with default names and structure.

Data source input

The **Import data** wizard connects to an external 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 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](#).

You should create this single table or view before running the wizard, and it must contain content. For obvious reasons, it doesn't make sense to run the **Import data** wizard on an empty data source.

SELECTION	DESCRIPTION
Existing data source	If you already have indexers defined in your search service, you might have an existing data source definition that you can reuse. In Azure Cognitive Search, data source objects are only used by indexers. You can create a data source object programmatically or through the Import data wizard, and reuse them as needed.
Samples	Azure Cognitive Search provides two built-in sample data sources that are used in tutorials and quickstarts: a real estate SQL database and a Hotels database hosted on Cosmos DB. For a walk through based on the Hotels sample, see the Create an index in the Azure portal quickstart.
Azure SQL Database or SQL Managed Instance	<p>Service name, credentials for a database user with read permission, and a database name can be specified either on the page or via an ADO.NET connection string. Choose the connection string option to view or customize properties.</p> <p>The table or view that provides the rowset must be specified on the page. This option appears after the connection succeeds, giving a drop-down list so that you can make a selection.</p>
SQL Server on Azure VM	<p>Specify a fully qualified service name, user ID and password, and database as a connection string. To use this data source, you must have previously installed a certificate in the local store that encrypts the connection. For instructions, see SQL VM connection to Azure Cognitive Search.</p> <p>The table or view that provides the rowset must be specified on the page. This option appears after the connection succeeds, giving a drop-down list so that you can make a selection.</p>
Azure Cosmos DB	Requirements include the account, database, and collection. All documents in the collection will be included in the index. You can define a query to flatten or filter the rowset, or leave the query blank. A query is not required in this wizard.
Azure Blob Storage	Requirements include the storage account and a container. Optionally, if blob names follow a virtual naming convention for grouping purposes, you can specify the virtual directory portion of the name as a folder under container. See Indexing Blob Storage for more information.

SELECTION	DESCRIPTION
Azure Table Storage	Requirements include the storage account and a table name. Optionally, you can specify a query to retrieve a subset of the tables. See Indexing Table Storage for more information.

Wizard output

Behind the scenes, the wizard creates, configures, and invokes the following objects. After the wizard runs, you can find its output in the portal pages. The Overview page of your service has lists of indexes, indexers, data sources, and skillsets. Index definitions can be viewed in full JSON in the portal. For other definitions, you can use the [REST API](#) to GET specific objects.

OBJECT	DESCRIPTION
Data Source	Persists connection information to source data, including credentials. A data source object is used exclusively with indexers.
Index	Physical data structure used for full text search and other queries.
Skillset	A complete set of instructions for manipulating, transforming, and shaping content, including analyzing and extracting information from image files. Except for very simple and limited structures, it includes a reference to a Cognitive Services resource that provides enrichment. Optionally, it might also contain a knowledge store definition.
Indexer	A configuration object specifying a data source, target index, an optional skillset, optional schedule, and optional configuration settings for error handling and base-64 encoding.

How to start the wizard

The Import data wizard is started from the command bar on the service Overview page.

1. In the [Azure portal](#), open the search service page from the dashboard or [find your service](#) in the service list.
2. In the service overview page at the top, click **Import data**.



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.

How to edit or finish an index schema in the wizard

The wizard generates an incomplete index, which will be populated with documents obtained from the input data source. For a functional index, make sure you have the following elements defined.

1. Is the field list complete? Add 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 is [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 will 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 does not appreciably effect 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, additional 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.

Next steps

The best way to understand the benefits and limitations of the wizard is to step through it. The following quickstart guides you through each step.

[Create an Azure Cognitive Search index using the Azure portal](#)

How to rebuild an index in Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

This article explains how to rebuild an Azure Cognitive Search index, the circumstances under which rebuilds are required, and recommendations for mitigating the impact of rebuilds on ongoing query requests.

A *rebuild* refers to dropping and recreating the physical data structures associated with an index, including all field-based inverted indexes. In Azure Cognitive Search, you cannot drop and recreate individual fields. To rebuild an index, all field storage must be deleted, recreated based on an existing or revised index schema, and then repopulated with data pushed to the index or pulled from external sources.

It's common to rebuild indexes during development when you are iterating over index design, but you might also need to rebuild a production-level index to accommodate structural changes, such as adding complex types or adding fields to suggesters.

"Rebuild" versus "refresh"

Rebuild should not be confused with refreshing the contents of an index with new, modified, or deleted documents. Refreshing a search corpus is almost a given in every search app, with some scenarios requiring up-to-the-minute updates (for example, when a search corpus needs to reflect inventory changes in an online sales app).

As long as you are not changing the structure of the index, you can refresh an index using the same techniques that you used to load the index initially:

- For push-mode indexing, call [Add, Update or Delete Documents](#) to push the changes to an index.
- For indexers, you can [schedule indexer execution](#) and use change-tracking or timestamps to identify the delta. If updates must be reflected faster than what a scheduler can manage, you can use push-mode indexing instead.

Rebuild conditions

Drop and recreate an index if any of the following conditions are true.

CONDITION	DESCRIPTION
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 <i>assign</i> 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 cannot delete or change an existing analyzer configuration (analyzer, tokenizer, token filter, or char filter) in the index unless you rebuild the entire index.

CONDITION	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.
Delete a field	To physically remove all traces of a field, you have to rebuild the index. When an immediate rebuild is not 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.
Switch tiers	If you require more capacity, there is no in-place upgrade in the Azure portal. A new service must be created, and indexes must be built from scratch on the new service. 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.

Update conditions

Many other modifications can be made without impacting existing physical structures. Specifically, the following changes do *not* require an index rebuild. For these changes, you can [update an index definition](#) with your changes.

- Add a new field
- Set the **retrievable** attribute on an existing field
- Set a **searchAnalyzer** on an existing field
- Add a new analyzer definition in an index
- Add, update, or delete scoring profiles
- Add, update, or delete CORS settings
- Add, update, or delete synonymMaps

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.

Indexing does not run in the background and the service will balance the additional indexing against ongoing queries. During indexing, you can [monitor query requests](#) in the portal to ensure queries are completing in a timely manner.

1. Determine whether a rebuild is required. If you are 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 are not 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 are not searchable can be used in filters or expressions, but do not have inverted indexes and are not 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.

NOTE

If you have stringent SLA requirements, you might consider provisioning a new service specifically for this work, with development and indexing occurring in full isolation from a production index. A separate service runs on its own hardware, eliminating any possibility of resource contention. When development is complete, you would either leave the new index in place, redirecting queries to the new endpoint and index, or you would run finished code to publish a revised index on your original Azure Cognitive Search service. There is currently no mechanism for moving a ready-to-use index to another service.

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](#)

How to index large data sets in Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

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.

As data volumes grow or processing needs change, you might find that simple or default indexing strategies are no longer practical. For Azure Cognitive Search, there are several approaches for accommodating larger data sets, ranging from how you structure a data upload request, to using a source-specific indexer for scheduled and distributed workloads.

The same techniques also apply to long-running processes. In particular, the steps outlined in [parallel indexing](#) are helpful for computationally intensive indexing, such as image analysis or natural language processing in an [AI enrichment pipeline](#).

The following sections explore techniques for indexing large amounts of data using both the push API and indexers.

Use the push API

When pushing data into an index using the [Add Documents REST API](#) or the [Index method](#), there are several key considerations that impact indexing speed. Those factors are outlined in the section below, and range from setting service capacity to code optimizations.

For more information and code samples that illustrate push model indexing, see [Tutorial: Optimize indexing speeds](#).

Capacity of your service

As a first step, review the characteristics and [limits](#) of the tier at which you provisioned the service. One of the key differentiating factors among the pricing tiers is the size and speed of partitions, which has a direct impact on indexing speed. If you provisioned your search service at a tier that is insufficient for the workload, upgrading to a new tier might be the easiest and most effective solution for increasing indexing throughput.

Once you are satisfied with the tier, your next step might be to increase the number of partitions. Partition allocation can be readjusted downwards after an initial indexing run to reduce the overall cost of running the service.

NOTE

Adding additional replicas may also increase indexing speeds but it isn't guaranteed. On the other hand, additional replicas will increase the query volume your search service can handle. Replicas are also a key component for getting an [SLA](#).

Before adding partition(replicas) or upgrading to a higher tier, consider the monetary cost and allocation time. Adding partitions can significantly increase indexing speed but adding/removing them can take anywhere from 15 minutes to several hours. For more information, see the documentation on [adjusting capacity](#).

Review index schema

The schema of your index plays an important role in indexing data. The more fields you have, and the more properties you set (such as *searchable*, *facetable*, or *filterable*) all contribute to increased indexing time. In general,

you should only create and specify the fields that you actually need in a search index.

NOTE

To keep document size down, avoid adding non-queryable data to an index. Images and other binary data are not directly searchable and shouldn't be stored in the index. To integrate non-queryable data into search results, you should define a non-searchable field that stores a URL reference to the resource.

Check the batch size

One of the simplest mechanisms for indexing a larger data set 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 [Index method](#) in the .NET SDK. For either API, you would package 1000 documents in the body of each request.

Using batches to index documents will significantly improve indexing performance. 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 what results in the fastest indexing speeds for your scenario. This [tutorial](#) provides sample code for testing batch sizes using the .NET SDK.

Number of 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.

The optimal number of threads is determined by:

- The tier of your search service
- The number of partitions
- The size of your batches
- The schema of your index

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.

NOTE

As you increase the tier of your search service or increase the partitions, you should also increase the number of concurrent threads.

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.

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.

Network data transfer speeds

Network data transfer speeds can be a limiting factor when indexing data. Indexing data from within your Azure environment is an easy way to speed up indexing.

Use indexers (pull API)

[Indexers](#) are used to crawl supported Azure data sources for searchable content. While not specifically intended for large-scale indexing, several indexer capabilities are particularly useful for accommodating larger data sets:

- Schedulers allow you to parcel out indexing at regular intervals so that you can spread it out over time.
- Scheduled indexing can resume at the last known stopping point. If a data source is not fully crawled within a 24-hour window, the indexer will resume indexing on day two at wherever it left off.
- Partitioning data into smaller individual data sources enables parallel processing. You can break up source data into smaller components, such as into multiple containers in Azure Blob storage, and then create corresponding, multiple [data source objects](#) in Azure Cognitive Search that can be indexed in parallel.

NOTE

Indexers are data-source-specific, so using an indexer approach is only viable for selected data sources on Azure: [SQL Database](#), [Blob storage](#), [Table storage](#), [Cosmos DB](#).

Check the `batchSize` argument on Create Indexer

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.

Scheduled indexing

Indexer scheduling is an important mechanism for processing large data sets, as well as slow-running processes like image analysis in a cognitive search pipeline. Indexer processing operates within a 24-hour window. If processing fails to finish within 24 hours, the behaviors of indexer scheduling can work to your advantage.

By design, scheduled indexing starts at specific intervals, with a job typically completing before resuming at the next scheduled interval. However, if processing does not complete within the interval, the indexer stops (because it ran out of time). At the next interval, processing resumes where it last left off, with the system keeping track of where that occurs.

In practical terms, for index loads spanning several days, you can put the indexer on a 24-hour schedule. When indexing resumes for the next 24-hour cycle, it restarts at the last known good document. In this way, an indexer can work its way through a document backlog over a series of days until all unprocessed documents are processed. For more information about setting schedules in general, see [Create Indexer REST API](#) or see [How to schedule indexers for Azure Cognitive Search](#).

Parallel indexing

A parallel indexing strategy is based on indexing multiple data sources in unison, where each data source definition specifies a subset of the data.

For non-routine, computationally intensive indexing requirements - such as OCR on scanned documents in a cognitive search pipeline, image analysis, or natural language processing - a parallel indexing strategy is often the right approach for completing a long-running process in the shortest time. If you can eliminate or reduce query requests, parallel indexing on a service that is not simultaneously handling queries is your best strategy option for working through a large body of slow-processing content.

Parallel processing has these elements:

- Subdivide source data among multiple containers or multiple virtual folders inside the same container.
- Map each mini data set to its own [data source](#), paired to its own [indexer](#).
- For cognitive search, reference the same [skillset](#) in each indexer definition.
- Write into the same target search index.
- Schedule all indexers to run at the same time.

NOTE

In Azure Cognitive Search, you cannot assign individual replicas or partitions to indexing or query processing. The system determines how resources are used. To understand the impact on query performance, you might try parallel indexing in a test environment before rolling it into production.

How to configure parallel indexing

For indexers, processing capacity is loosely based on one indexer subsystem for each service unit (SU) used by your search service. Multiple concurrent indexers are possible on Azure Cognitive Search services provisioned on Basic or Standard tiers having at least two replicas.

1. In the [Azure portal](#), on your search service dashboard **Overview** page, check the **Pricing tier** to confirm it can accommodate parallel indexing. Both Basic and Standard tiers offer multiple replicas.
2. You can run as many indexers in parallel as the number of search units in your service. In **Settings > Scale**, [increase replicas](#) or partitions for parallel processing: one additional replica or partition for each indexer workload. Leave a sufficient number for existing query volume. Sacrificing query workloads for indexing is not a good tradeoff.
3. Distribute data into multiple containers at a level that Azure Cognitive Search indexers can reach. This could be multiple tables in Azure SQL Database, multiple containers in Azure Blob storage, or multiple collections. Define one data source object for each table or container.
4. Create and schedule multiple indexers to run in parallel:
 - Assume a service with six replicas. Configure six indexers, each one mapped to a data source containing one-sixth of the data set for a 6-way split of the entire data set.
 - Point each indexer to the same index. For cognitive search workloads, point each indexer to the same skillset.
 - Within each indexer definition, schedule the same run-time execution pattern. For example, `"schedule" : { "interval" : "PT8H", "startTime" : "2018-05-15T00:00:00Z" }` creates a schedule on 2018-05-15 on all indexers, running at eight-hour intervals.

At the scheduled time, all indexers begin execution, loading data, applying enrichments (if you configured a cognitive search pipeline), and writing to the index. Azure Cognitive Search does not lock the index for updates. Concurrent writes are managed, with retry if a particular write does not succeed on first attempt.

NOTE

When increasing replicas, consider increasing the partition count if index size is projected to increase significantly. Partitions store slices of indexed content; the more partitions you have, the smaller the slice each one has to store.

See also

- [Indexer overview](#)
- [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](#)

How to manage concurrency in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

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 craft 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](#) is probably 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, datasources, 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. Any object inheriting from [IResourceWithETag \(.NET SDK\)](#) has 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

```
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.

```
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

Review the [synonyms C# sample](#) for more context on how to safely update an existing index.

Try modifying either of the following samples to include ETags or AccessCondition objects.

- [REST API sample on GitHub](#)
- [.NET SDK sample on GitHub](#). This solution includes the "DotNetEtagsExplainer" project containing the code presented in this article.

See also

[Common HTTP request and response headers](#) [HTTP status codes](#) [Index operations \(REST API\)](#)

How to monitor Azure Cognitive Search indexer status and results

10/4/2020 • 5 minutes to read • [Edit Online](#)

Azure Cognitive Search provides status and monitoring information about current and historical runs of every indexer.

Indexer monitoring is useful when you want to:

- Track the progress of an indexer during an ongoing run.
- Review the results of ongoing or previous indexer run.
- Identify top-level indexer errors, and errors or warnings about individual documents being indexed.

Get status and history

You can access indexer monitoring information in various ways, including:

- In the [Azure portal](#)
- Using the [REST API](#)
- Using the [.NET SDK](#)

Available indexer monitoring information includes all the following (though the data formats differ based on the access method used):

- Status information about the indexer itself
- Information about the most recent run of the indexer, including its status, start and end times, and detailed errors and warnings.
- A list of historical indexer runs, and their statuses, results, errors, and warnings.

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. 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.

Monitor using the portal

You can see the current status of all of your indexers in the **Indexes** list on your search service Overview page.

Usage	Monitoring	Indexes	<u>Indexers</u>	Data sources	Skillsets
Last result		Name	Status	Last run	Docs succeeded
?		azure-sql-indexer	In progress	Just now	0/0
!		hotel-rooms-blob-indexer	Failed	6 min ago	0/1
i		hotel-rooms-cosmos-indexer	Reset	4 min ago	7/7
✓		myindexer	Success	8 min ago	50/50

When an indexer is executing, the status in the list shows **In Progress**, and the **Docs Succeeded** value shows the number of documents processed so far. It can take a few minutes for the portal to update indexer status values and

document counts.

An indexer whose most recent run was successful shows **Success**. 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.

If the most recent run ended with an error, the status shows **Failed**. A status of **Reset** means that the indexer's change tracking state was reset.

Click on an indexer in the list to see more details about the indexer's current and recent runs.

hotel-rooms-blob-indexer

Indexer

Run Reset Edit Delete

Indexer summary

Execution history

SUCCEEDED
FAILED

Execution details

Last Result	Last Run	Status	Docs Succeeded
!	6/28, 03:29 UTC	Failed	1/2
!	6/28, 03:28 UTC	Reset	0/0
✓	6/28, 00:00 UTC	Success	0/0
✓	6/27, 22:02 UTC	Success	7/7
!	6/27, 22:02 UTC	Reset	0/0
✓	6/27, 22:00 UTC	Success	7/7
!	6/27, 22:00 UTC	Reset	0/0

✓	6/27, 21:23 UTC	Success	7/7
ℹ	6/27, 21:23 UTC	Reset	0/0
✓	6/27, 21:14 UTC	Success	7/7
1 2 3 4 5 < >			

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.

Click 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.

myindexer

Execution

Execution result

0 Errors/Warnings ✓

Status	Success
Datasource	test-cosmos-src
Target index	test-cosmosdb-index
Start	6/28, 03:27 UTC
End	6/28, 03:27 UTC
Documents succeeded	50
Documents failed	0

Errors

[]

If there were document-specific problems during the run, they will be listed in the Errors and Warnings fields.

hotel-rooms-blob-indexer



Execution

Execution result

1 Errors/Warnings !

Status	Failed
Datasource	sampleblobstore
Target index	hotel-rooms-sample
Start	6/28, 03:29 UTC
End	6/28, 03:29 UTC
Documents succeeded	0
Documents failed	1

Errors

```
[{"error": { "key": "https://sampleblobstore.blob.core.windows.net/hotel-rooms/Rooms11.json", "errorMessage": "Document key cannot be missing or empty.\r\n"}]}
```

Warnings

```
[ ]
```

Warnings are common with some types of indexers, and do not 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 [Troubleshooting common indexer issues in Azure Cognitive Search](#).

Monitor using REST APIs

You can retrieve the status and execution history of an indexer using the [Get Indexer Status command](#):

```
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.

```
{  
    "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. A 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 details about status codes and indexer monitoring data, see [GetIndexerStatus](#).

Monitor using the .NET SDK

You can define the schedule for an indexer using the Azure Cognitive Search .NET SDK. To do this, include the **schedule** property when creating or updating an Indexer.

The following C# example writes information about an indexer's status and the results of its most recent (or ongoing) run to the console.

```

static void CheckIndexerStatus(Indexer indexer, SearchServiceClient searchService)
{
    try
    {
        IndexerExecutionInfo execInfo = searchService.Indexers.GetStatus(indexer.Name);

        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:

```

Indexer has run 18 times.
Indexer Status: Running
Latest run
Run Status: Success
Total Documents: 7, Failed: 0
StartTime: 10:02:46 PM, EndTime: 10:02:47 PM, Elapsed: 00:00:01.0990000
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. A indexer status of **Running** means that the indexer is set up correctly and available for execution, but not that it is 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.

For more details about status codes and indexer monitoring information, see [GetIndexerStatus](#) in the REST API.

Details about document-specific errors or warnings can be retrieved by enumerating the lists

`IndexerExecutionResult.Errors` and `IndexerExecutionResult.Warnings`.

For more information about the .NET SDK classes used to monitor indexers, see [IndexerExecutionInfo](#) and [IndexerExecutionResult](#).

How to schedule indexers in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

An indexer normally runs once, immediately after it is created. You can run it again on demand using the portal, the REST API, or the .NET SDK. You can also configure an indexer to run periodically on a schedule.

Some situations where indexer scheduling is useful:

- Source data will change over time, and you want the Azure Cognitive Search indexers to automatically process the changed data.
- The index will be populated from multiple data sources and you want to make sure the indexers run at different times to reduce conflicts.
- The source data is very large and you want to spread the indexer processing over time. For more information about indexing large volumes of data, see [How to index large data sets in Azure Cognitive Search](#).

The scheduler is a built-in feature of Azure Cognitive Search. You can't use an external scheduler to control search indexers.

Define schedule properties

An indexer schedule has two properties:

- **Interval**, which defines the amount of time in between scheduled indexer executions. The smallest interval allowed is 5 minutes, and the largest is 24 hours.
- **Start Time (UTC)**, which indicates the first time at which the indexer should be run.

You can specify a schedule when first creating the indexer, or by updating the indexer's properties later. Indexer schedules can be set using the [portal](#), the [REST API](#), or the [.NET SDK](#).

Only one execution of an indexer can run at a time. If an indexer is already running when its next execution is scheduled, that execution is postponed until the next scheduled time.

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, 2019 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, 2019 at 8:00 AM UTC. Assume this execution takes 20 minutes (or any time less than 1 hour).
- The second execution starts at or around June 1, 2019 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 will not start until 11:00 AM UTC.

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.

Schedule in the portal

The Import Data wizard in the portal lets you define the schedule for an indexer at creation time. The default Schedule setting is **Hourly**, which means the indexer runs once after it is created, and runs again every hour afterwards.

You can change the Schedule setting to **Once** if you don't want the indexer to run again automatically, or to **Daily** to run once per day. Set it to **Custom** if you want to specify a different interval or a specific future Start Time.

When you set the schedule to **Custom**, fields appear to let you specify the **Interval** and the **Start Time (UTC)**. The shortest time interval allowed is 5 minutes, and the longest is 1440 minutes (24 hours).

Import data

* Connect to your data Add cognitive search (Optional) * Customize target index *** Create an indexer**

Indexer

* Name: myindexer ✓

Schedule: **Custom** (Once, Hourly, Daily, Custom) Custom

* Interval (minutes): 240 ✓

* Start time (UTC): 2019-06-01 8:00:00 AM

Change tracking automatically configured with a high watermark policy.

Track deletions:

Description: (optional)

Advanced options

[Previous: Customize target index](#) **Submit**

After an indexer has been created, you can change the schedule settings using the indexer's Edit panel. The Schedule fields are the same as in the Import Data wizard.

Edit X

Index
test-cosmosdb-index ▼

Data Source
test-cosmos-src ▼

Schedule i
Once Hourly Daily Custom

* Interval (minutes)
240

* Start time (UTC) i
2019-06-01 8:00:00 AM

Advanced options >

Description

OK

Schedule using REST APIs

You can define the schedule for an indexer using the REST API. To do this, include the `schedule` property when creating or updating the indexer. The example below shows a PUT request to update an existing indexer:

```

PUT https://myservice.search.windows.net/indexers/myindexer?api-version=2020-06-30
Content-Type: application/json
api-key: admin-key

{
    "dataSourceName" : "myazuresqldatasource",
    "targetIndexName" : "target index name",
    "schedule" : { "interval" : "PT10M", "startTime" : "2015-01-01T00:00:00Z" }
}

```

The **interval** parameter is required. The interval refers to the time between the start of two consecutive indexer executions. The smallest allowed interval is 5 minutes; the longest is one day. 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 2 hours.

The optional **startTime** indicates when scheduled executions should begin. If it is omitted, the current UTC 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 **startTime**.

You can also run an indexer on demand at any time using the [Run Indexer](#) call. For more information about running indexers and setting indexer schedules, see [Run Indexer](#), [Get Indexer](#), and [Update Indexer](#) in the REST API Reference.

Schedule using the .NET SDK

You can define the schedule for an indexer using the Azure Cognitive Search .NET SDK. To do this, include the **schedule** property when creating or updating an Indexer.

The following C# example creates an indexer, using a predefined data source and index, and sets its schedule to run once every day starting 30 minutes from now:

```

Indexer indexer = new Indexer(
    name: "azure-sql-indexer",
    dataSourceName: dataSource.Name,
    targetIndexName: index.Name,
    schedule: new IndexingSchedule(
        TimeSpan.FromDays(1),
        new DateTimeOffset(DateTime.UtcNow.AddMinutes(30))
    )
);
await searchService.Indexers.CreateOrUpdateAsync(indexer);

```

If the **schedule** parameter is omitted, the indexer will only run once immediately after it is created.

The **startTime** parameter can be set to a time in the past. In that case, the first execution is scheduled as if the indexer has been running continuously since the given **startTime**.

The schedule is defined using the [IndexingSchedule](#) class. The [IndexingSchedule](#) constructor requires an **interval** parameter specified using a [TimeSpan](#) object. The smallest interval value allowed is 5 minutes, and the largest is 24 hours. The second **startTime** parameter, specified as a [DateTimeOffset](#) object, is optional.

The .NET SDK lets you control indexer operations using the [SearchServiceClient](#) class and its [Indexers](#) property, which implements methods from the [IIndexersOperations](#) interface.

You can run an indexer on demand at any time using one of the [Run](#), [RunAsync](#), or [RunWithHttpMessagesAsync](#) methods.

For more information about creating, updating, and running indexers, see [IIndexersOperations](#).

Field mappings and transformations using Azure Cognitive Search indexers

10/4/2020 • 9 minutes to read • [Edit Online](#)



When using Azure Cognitive Search indexers, you sometimes find that the input data doesn't quite match the schema of your target index. In those cases, you can use **field mappings** to reshape your data during the indexing process.

Some situations where field mappings are useful:

- Your data source has a field named `_id`, but Azure Cognitive Search doesn't allow field names that start with an underscore. A field mapping lets you effectively rename a field.
- You want to populate several fields in the index from the same data source data. For example, you might want to apply different analyzers to those fields.
- You want to populate an index field with data from more than one data source, and the data sources each use different field names.
- You need to Base64 encode or decode your data. Field mappings support several **mapping functions**, including functions for Base64 encoding and decoding.

NOTE

Field mappings in indexers are a simple way to map data fields to index fields, with some ability for light-weight data conversion. More complex data might require pre-processing to reshape it into a form that's conducive to indexing. One option you might consider is [Azure Data Factory](#).

Set up field mappings

A field mapping consists of three parts:

1. A `sourceFieldName`, which represents a field in your data source. This property is required.
2. An optional `targetFieldName`, which represents a field in your search index. If omitted, the same name as in the data source is used.
3. An optional `mappingFunction`, which can transform your data using one of several predefined functions. This can be applied on both input and output field mappings. The full list of functions is [below](#).

Field mappings are added to the `fieldMappings` array of the indexer definition.

NOTE

If no field mappings are added, indexers assume data source fields should be mapped to index fields with the same name. Adding a field mapping removes these 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.

Map fields using the REST API

You can add field mappings when creating a new indexer using the [Create Indexer](#) API request. You can manage the field mappings of an existing indexer using the [Update Indexer](#) API request.

For example, here's how to map a source field to a target field with a different name:

```
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" : "_id", "targetFieldName" : "id" } ]
}
```

A source field can be referenced in multiple field mappings. The following example shows how to "fork" a field, copying the same source field to two different index fields:

```
"fieldMappings" : [
    { "sourceFieldName" : "text", "targetFieldName" : "textStandardEnglishAnalyzer" },
    { "sourceFieldName" : "text", "targetFieldName" : "textSoundexAnalyzer" }
]
```

NOTE

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 cannot have fields that differ only by case.

Map fields using the .NET SDK

You define field mappings in the .NET SDK using the [FieldMapping](#) class, which has the properties `SourceFieldName` and `TargetFieldName`, and an optional `MappingFunction` reference.

You can specify field mappings when constructing the indexer, or later by directly setting the `Indexer.FieldMappings` property.

The following C# example sets the field mappings when constructing an indexer.

```
List<FieldMapping> map = new List<FieldMapping> {
    // removes a leading underscore from a field name
    new FieldMapping("_custId", "custId"),
    // URL-encodes a field for use as the index key
    new FieldMapping("docPath", "docId", FieldMappingFunction.Base64Encode() )
};

Indexer sqlIndexer = new Indexer(
    name: "azure-sql-indexer",
    dataSourceName: sqlDataSource.Name,
    targetIndexName: index.Name,
    fieldMappings: map,
    schedule: new IndexingSchedule(TimeSpan.FromDays(1)));

await searchService.Indexers.CreateOrUpdateAsync(indexer);
```

Field mapping functions

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 - document key lookup

Only URL-safe characters can appear in an Azure Cognitive Search document key (because customers must be able to address the document using the [Lookup API](#)). If the source field for your key contains URL-unsafe characters, you can use the `base64Encode` function to convert it at indexing time. However, 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, you can then use the `base64Decode` function to get the original key value, and use that to retrieve the source document.

```
"fieldMappings" : [
  {
    "sourceFieldName" : "SourceKey",
    "targetFieldName" : "IndexKey",
    "mappingFunction" : {
      "name" : "base64Encode",
      "parameters" : { "useHttpServerUtilityUrlTokenEncode" : false }
    }
  }
]
```

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:

```
"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.

```
"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 details, 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 \(Azure Cognitive Search\)](#) 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 leverage this built-in functionality. If you are 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 additional 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	ADDITIONAL PROCESSING AFTER LIBRARY ENCODING	ADDITIONAL 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.

```
"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 cannot 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.

```
"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 are not 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.

```
"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.

```
"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 facing errors complaining about document key being longer than 1024 characters, this function can be applied to reduce the length of the document key.

```
"fieldMappings" : [
  {
    "sourceFieldName" : "metadata_storage_path",
    "targetFieldName" : "your key field",
    "mappingFunction" : {
      "name" : "fixedLengthEncode"
    }
  }
]
```

Set up an indexer connection to a data source using a managed identity

10/4/2020 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Setting up a connection to a data source using a managed identity is not supported with the Free Azure Cognitive Search tier.

An [indexer](#) in Azure Cognitive Search is a crawler that provides a way to pull data from your data source into Azure Cognitive Search. An indexer obtains a data source connection from the data source object that you create. The data source object usually includes credentials for the target data source. For example, the data source object could include an Azure Storage account key if you want to index data from a blob storage container.

In many cases providing credentials directly in the data source object is not a problem, but there are some challenges that can come up:

- How do I keep the credentials secure in my code that creates the data source object?
- If my account key or password is compromised and I need to change it, I now need to update my data source objects with the new account key or password so that my indexer can connect to the data source again.

These concerns can be resolved by setting up your connection using a managed identity.

Using managed identities

[Managed identities](#) is a feature that provides Azure services with an automatically managed identity in Azure Active Directory (Azure AD). You can use this feature in Azure Cognitive Search to create a data source object with a connection string that does not include any credentials. Instead, your search service will be granted access to the data source through role-based access control (RBAC).

When setting up a data source using a managed identity, you can change your data source credentials and your indexers will still be able to connect to the data source. You can also create data source objects in your code without having to include an account key or use Key Vault to retrieve an account key.

Limitations

The following data sources support setting up an indexer connection using managed identities.

- [Azure Blob storage](#), [Azure Data Lake Storage Gen2 \(preview\)](#), [Azure Table storage](#)
- [Azure Cosmos DB](#)
- [Azure SQL Database](#)

The following features do not currently support using managed identities to set up the connection:

- Knowledge Store
- Custom skills

Next steps

Learn more about how to set up an indexer connection using managed identities:

- [Azure Blob storage](#), [Azure Data Lake Storage Gen2 \(preview\)](#), [Azure Table storage](#)
- [Azure Cosmos DB](#)
- [Azure SQL Database](#)

Set up a connection to an Azure Storage account using a managed identity

10/4/2020 • 4 minutes to read • [Edit Online](#)

This page describes how to set up an indexer connection to an Azure storage account using a managed identity instead of providing credentials in the data source object connection string.

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](#)
- [Azure Blob indexer](#)
- [Azure Data Lake Storage Gen2 indexer](#)
- [Azure Table indexer](#)

Set up the connection

1 - Turn on system-assigned 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 role-based access control (RBAC) assignments that allow access to data during indexing.

The screenshot shows the Azure portal interface for managing the identity of a cognitive search service named "azure-cognitive-search-service". The left sidebar lists various service settings like Overview, Activity log, Access control (IAM), Tags, and Identity. The "Identity" option is highlighted with a red box. The main content area shows the "System assigned" tab selected, with a descriptive text about system-assigned identities. Below the text are "Save", "Discard", "Refresh", and "Got feedback?" buttons, with "Save" being highlighted by a red box. At the bottom, there's a "Status" switch with "Off" and "On" options, where "On" is highlighted by a red box.

After selecting **Save** you will see an Object ID that has been assigned to your search service.

The screenshot shows the 'Identity' settings page for an Azure Cognitive Search service. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick start, Keys, Scale, Search traffic analytics, Identity (which is selected and highlighted in grey), Properties, and a 'More' section. The main pane has a header 'System assigned' with a description: 'A system assigned managed identity enables Azure resources to authenticate to cloud services (e.g. Azure Key Vault)'. It features a 'Save' button, a 'Discard' button, a 'Refresh' button, and a 'Got feedback?' link. Below this is a 'Status' switch set to 'On'. An 'Object ID' input field contains the value '1da4d008-e5ce-4a03-8041-4694e5059d36', which is also copied to the clipboard. A 'Permissions' section lists 'Azure role assignments'. A note at the bottom states: 'This resource is registered with Azure Active Directory. You can control its access to services like Azure Resource Manager and other Azure services.' A red box highlights the 'Object ID' input field.

2 - Add a role assignment

In this step you will give your Azure Cognitive Search service permission to read data from your storage account.

1. In the Azure portal, navigate to the Storage account that contains the data that you would like to index.
2. Select **Access control (IAM)**
3. Select **Add** then **Add role assignment**

The screenshot shows the 'Access control (IAM)' settings page for a storage account. The left sidebar includes Overview, Activity log, Access control (IAM) (selected and highlighted in grey), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), Settings, Access keys, Geo-replication, CORS, Configuration, and Encryption. The main pane has a header with a '+ Add' button, 'Edit columns', 'Refresh', 'Remove', and 'Got feedback?' links. It shows sections for 'Add role assignment', 'Add co-administrator', 'Deny assignments', 'Classic administrators', and 'Roles'. A 'Check access' section allows reviewing the level of access for users, groups, service principals, or managed identities. A 'Find' dropdown is set to 'Azure AD user, group, or service principal'. A 'Search by name or email address' input field is present. To the right, there are two sections: 'Add a role assignment' (with a checked checkbox and a 'Add' button) and 'View deny assignments' (with a 'View' button). A red box highlights the '+ Add' button in the header.

4. Select the appropriate role(s) based on the storage account type that you would like to index:
 - a. Azure Blob storage requires that you add your search service to the **Storage Blob Data Reader** role.
 - b. Azure Data Lake Storage Gen2 requires that you add your search service to the **Storage Blob Data Reader** role.
 - c. Azure Table storage requires that you add your search service to the **Reader and Data Access** role.
5. Leave **Assign access to** as **Azure AD user, group or service principal**
6. Search for your search service, select it, then select **Save**

Example for Azure Blob storage and Azure Data Lake Storage Gen2:

The screenshot shows the 'Access control (IAM)' blade for a Microsoft StorageAccount. On the left, there's a sidebar with various settings like Overview, Activity log, and Access control (IAM). The main area has tabs for 'Check access', 'Role assignments', 'Deny assignments', 'Classic administrators', and 'Roles'. Under 'Role assignments', a 'Storage Blob Data Reader' role is selected. The 'Selected members' section shows 'azure-cognitive-search-service' with a 'Remove' button. At the bottom are 'Save' and 'Discard' buttons.

Example for Azure Table storage:

This screenshot is identical to the one above, showing the 'Access control (IAM)' blade for a Microsoft StorageAccount. It displays the same interface for assigning roles like 'Reader and Data Access' to service principals, with a selected member 'azure-cognitive-search-service' and save/discard buttons at the bottom.

3 - Create the data source

The [REST API](#), Azure portal, and the [.NET SDK](#) support the managed identity connection string. Below is an example of how to create a data source to index data from a storage account using the [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.

When indexing from a storage account, the data source must have the following required properties:

- **name** is the unique name of the data source within your search service.
- **type**
 - Azure Blob storage: `azureblob`
 - Azure Table storage: `azuretable`
 - Azure Data Lake Storage Gen2: **type** will be provided once you sign up for the preview using [this form](#).
- **credentials**
 - When using a managed identity to authenticate, the **credentials** format is different than when not using a managed identity. Here you will provide a Resourceld that has no account key or password. The Resourceld must include the subscription ID of the storage account, the resource group of the storage account, and the storage account name.
 - Managed identity format:

- *ResourceId=/subscriptions/your subscription ID/resourceGroups/your resource group name/providers/Microsoft.Storage/storageAccounts/your storage account name/;*
- **container** specifies a container or table name in your storage account. By default, all blobs within the container are retrievable. If you only want to index blobs in a particular virtual directory, you can specify that directory using the optional **query** parameter.

Example of how to create a blob data source object using the [REST API](#):

```
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/00000000-0000-0000-0000-000000000000/resourceGroups/resource-group-name/providers/Microsoft.Storage/storageAccounts/storage-account-name/;" },
  "container" : { "name" : "my-container", "query" : "<optional-virtual-directory-name>" }
}
```

4 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's how to create an index with a searchable `content` field to store the text extracted from blobs:

```
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 }
  ]
}
```

For more on creating indexes, see [Create Index](#)

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.

Example indexer definition for a blob indexer:

```
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",
  "schedule" : { "interval" : "PT2H" }
}
```

This indexer will run every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out [Create Indexer](#).

For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

Accessing secure data in storage accounts

Azure storage accounts can be further secured using firewalls and virtual networks. If you want to index content from a blob storage account or Data Lake Gen2 storage account that is secured using a firewall or virtual network, follow the instructions for [Accessing data in storage accounts securely via trusted service exception](#).

See also

Learn more about Azure Storage indexers:

- [Azure Blob indexer](#)
- [Azure Data Lake Storage Gen2 indexer](#)
- [Azure Table indexer](#)

Set up an indexer connection to a Cosmos DB database using a managed identity

10/4/2020 • 4 minutes to read • [Edit Online](#)

This page describes how to set up an indexer connection to an Azure Cosmos DB database using a managed identity instead of providing credentials in the data source object connection string.

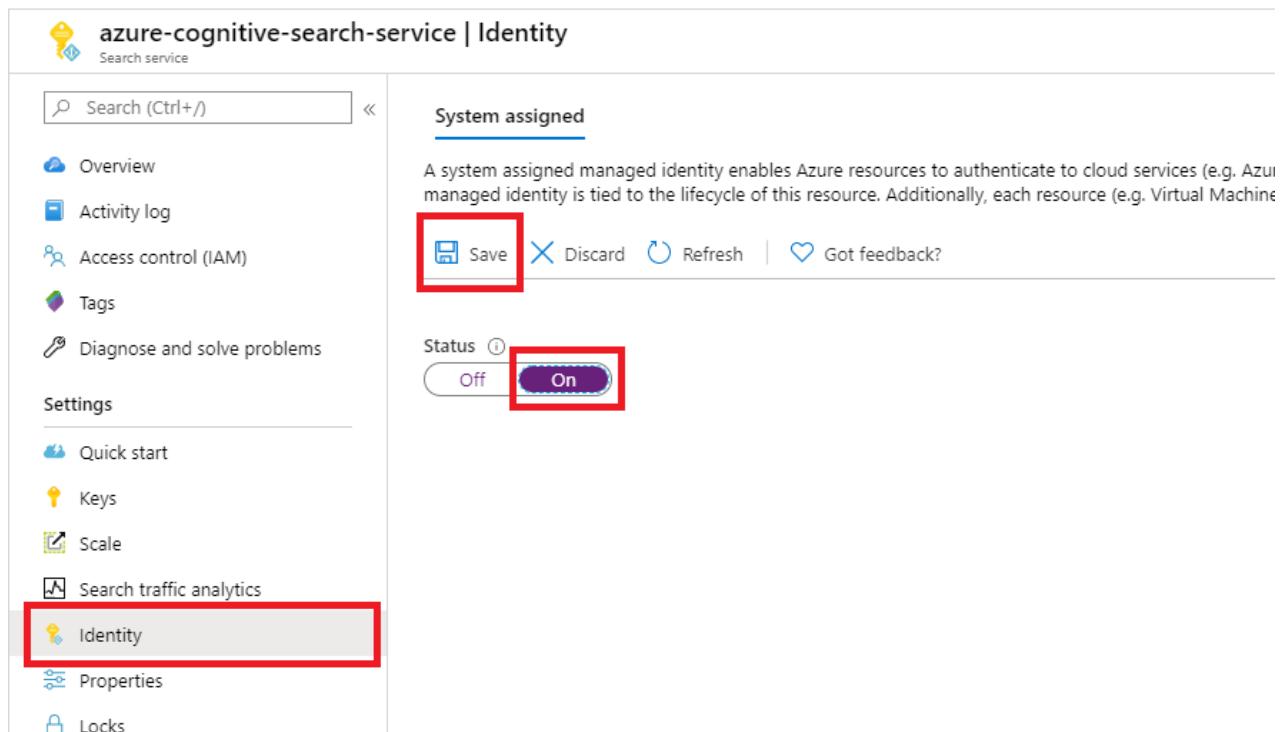
Before learning more about this feature, it is recommended that you have an understanding of what an indexer is and how to set up an indexer for your data source. More information can be found at the following links:

- [Indexer overview](#)
- [Azure Cosmos DB indexer](#)

Set up a connection using a managed identity

1 - Turn on system-assigned 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 role-based access control (RBAC) assignments that allow access to data during indexing.



The screenshot shows the Azure portal interface for managing the identity of an Azure Cognitive Search service named "azure-cognitive-search-service". The left sidebar lists various service settings like Overview, Activity log, Access control (IAM), Tags, and Identity. The "Identity" section is highlighted with a red box. The main content area shows the "System assigned" tab selected under "Identity". A descriptive text explains that a system-assigned managed identity enables Azure resources to authenticate to cloud services. Below this, there are "Save", "Discard", "Refresh", and "Got feedback?" buttons. A "Status" switch is shown as "On", also highlighted with a red box. The overall theme is light blue and white, typical of the Azure UI.

After selecting **Save** you will see an Object ID that has been assigned to your search service.

The screenshot shows the 'Identity' settings page for the 'azure-cognitive-search-service'. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick start, Keys, Scale, Search traffic analytics, and Identity (which is selected). The main area shows a 'System assigned' identity with the status set to 'On'. The 'Object ID' field contains the value '1da4d008-e5ce-4a03-8041-4694e5059d36', which is highlighted with a red box. Below it is a 'Permissions' section with a 'Azure role assignments' button. A note at the bottom states: 'This resource is registered with Azure Active Directory. You can control its access to services like Azure Resource Manager and other Azure services using roles defined in your directory.'

2 - Add a role assignment

In this step you will give your Azure Cognitive Search service permission to read data from your Cosmos DB database.

1. In the Azure portal, navigate to the Cosmos DB account that contains the data that you would like to index.
2. Select **Access control (IAM)**
3. Select **Add** then **Add role assignment**

The screenshot shows the 'Access control (IAM)' settings page for the 'my-cosmos-db-account'. The left sidebar includes options like Overview, Activity log, Access control (IAM) (which is selected and highlighted with a red box), Tags, Diagnose and solve problems, Quick start, Notifications, Data Explorer, and Settings. The main area shows a 'Check access' section with a 'Find' dropdown set to 'Azure AD user, group, or service principal' and a search bar. To the right, there are two sections: 'Add a role assignment' (with a 'Add' button) and 'View deny as' (with a 'View' button).

4. Select the **Cosmos DB Account Reader Role**
5. Leave **Assign access to** as **Azure AD user, group or service principal**
6. Search for your search service, select it, then select **Save**

3 - Create the data source

The [REST API](#), Azure portal, and the [.NET SDK](#) support the managed identity connection string. Below is an example of how to create a data source to index data from Cosmos DB using the [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.

When using managed identities to authenticate, the **credentials** will not include an account key.

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "cosmos-db-datasource",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "Database=sq1-test-db;ResourceId=/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/cosmos-db-resource-group/providers/Microsoft.DocumentDB/databaseAccounts/my-cosmos-db-account/";
    },
    "container": { "name": "myCollection", "query": null },
    "dataChangeDetectionPolicy": {
        "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "_ts"
    }
}
```

The body of the request contains the data source definition, which should include the following fields:

FIELD	DESCRIPTION
name	Required. Choose any name to represent your data source object.
type	Required. Must be <code>cosmosdb</code> .

FIELD	DESCRIPTION
credentials	<p>Required.</p> <p>When connecting using a managed identity, the credentials format should be: <i>Database=[database-name];ResourceId=[resource-id-string];(ApiKind=[api-kind]);</i></p> <p>The ResourceId format: <i>ResourceId=/subscriptions/your subscription ID/resourceGroups/your resource group name/providers/Microsoft.DocumentDB/databaseAccounts/your cosmos db account name/</i></p> <p>For SQL collections, the connection string does not require an ApiKind.</p> <p>For MongoDB collections, add ApiKind=MongoDb to the connection string.</p> <p>For Gremlin graphs and Cassandra tables, sign up for the gated indexer preview to get access to the preview and information about how to format the credentials.</p>
container	<p>Contains the following elements:</p> <p>name: Required. Specify the ID of the database collection to be indexed.</p> <p>query: Optional. You can specify a query to flatten an arbitrary JSON document into a flat schema that Azure Cognitive Search can index.</p> <p>For the MongoDB API, Gremlin API, and Cassandra API, queries are not supported.</p>
dataChangeDetectionPolicy	Recommended
dataDeletionDetectionPolicy	Optional

4 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's how to create an index 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 }
  ]
}
```

For more on creating indexes, see [Create Index](#)

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.

Example indexer definition:

```
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",
  "schedule" : { "interval" : "PT2H" }
}
```

This indexer will run every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out [Create Indexer](#).

For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

Troubleshooting

If you find that you are not able to index data from Cosmos DB consider the following:

1. If you recently rotated your Cosmos DB account keys you will need to wait up to 15 minutes for the managed identity connection string to work.
2. Check to see if the Cosmos DB account has its access restricted to select networks. If it does, refer to [Indexer access to data sources using Azure network security features](#).

See also

Learn more about Cosmos DB indexers:

- [Azure Cosmos DB indexer](#)

Set up an indexer connection to Azure SQL Database using a managed identity

10/4/2020 • 4 minutes to read • [Edit Online](#)

This page describes how to set up an indexer connection to Azure SQL Database using a managed identity instead of providing credentials in the data source object connection string.

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](#)
- [Azure SQL indexer](#)

Set up a connection using a managed identity

1 - Turn on system-assigned 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 role-based access control (RBAC) assignments that allow access to data during indexing.

The screenshot shows the Azure portal interface for managing the identity of an Azure Cognitive Search service named "azure-cognitive-search-service". The left sidebar lists various service settings like Overview, Activity log, Access control (IAM), Tags, and Identity. The "Identity" option is highlighted with a red box. The main content area shows the "System assigned" tab selected under "Identity". A descriptive text explains that a system-assigned managed identity enables authentication to cloud services. Below this, there are "Save", "Discard", "Refresh", and "Got feedback?" buttons. At the bottom, a "Status" switch is shown, with the "On" button highlighted by a red box. The "Off" button is also visible.

After selecting **Save** you will see an Object ID that has been assigned to your search service.

The screenshot shows the 'Identity' settings for an Azure Cognitive Search service. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Quick start, Keys, Scale, Search traffic analytics, and Identity (which is selected). The main area has a heading 'System assigned' and a status switch set to 'On'. Below that is an 'Object ID' input field containing '1da4d008-e5ce-4a03-8041-4694e5059d36', which is also highlighted with a red box. A note below says 'This resource is registered with Azure Active Directory. You can control its access to services like Azure Resource Manager'. At the top right are Save, Discard, Refresh, and Got feedback? buttons.

2 - Provision Azure Active Directory Admin for SQL Server

When connecting to the database in the next step, you will need to connect with an Azure Active Directory (Azure AD) account that has admin access to the database in order to give your search service permission to access the database.

Follow the instructions [here](#) to give your Azure AD account admin access to the database.

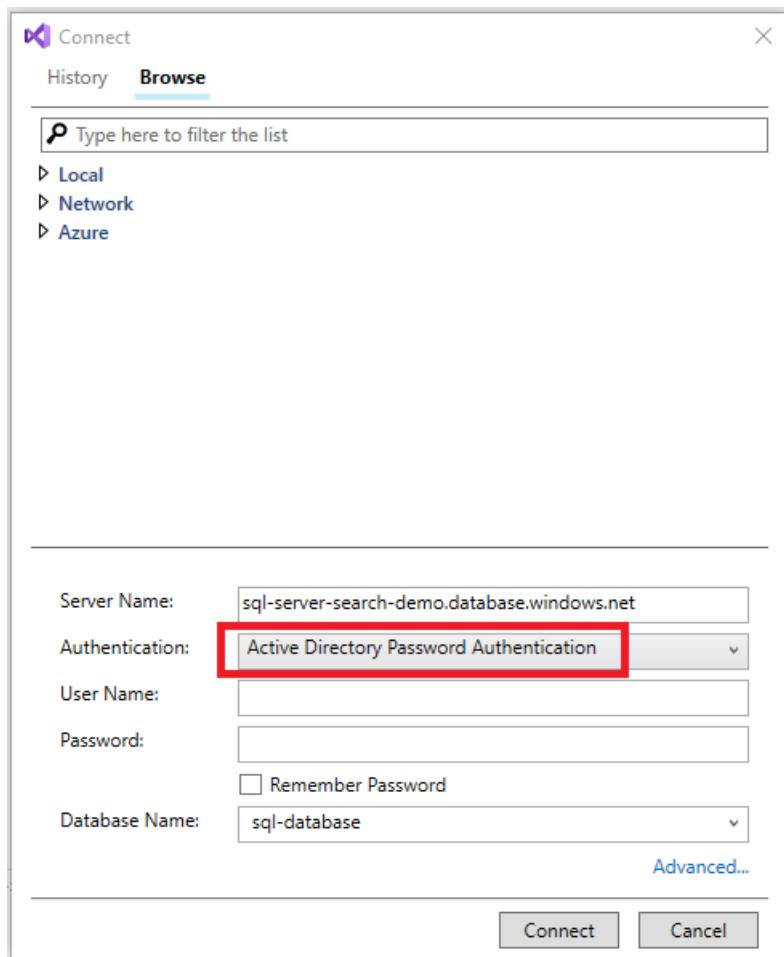
3 - Assign the search service permissions

Follow the below steps to assign the search service permission to read the database.

1. Connect to Visual Studio

The screenshot shows the 'sql-database' settings page for an Azure SQL database. The left sidebar includes Overview, Activity log, Tags, Diagnose and solve problems, Quick start, Query editor (preview), Power Platform (Power BI, Power Apps, Power Automate), and Settings (Configure, Geo-Replication, Connection strings, Sync to other databases). The main area displays database details: Resource group (change), Status: Online, Location: East US, Subscription (change), Subscription ID, and Tags (change). It also shows Compute utilization with a scale from 10% to 100%. At the top right, there are buttons for Copy, Restore, Export, Set server firewall, Delete, Connect with... (highlighted with a red box), and Feedback.

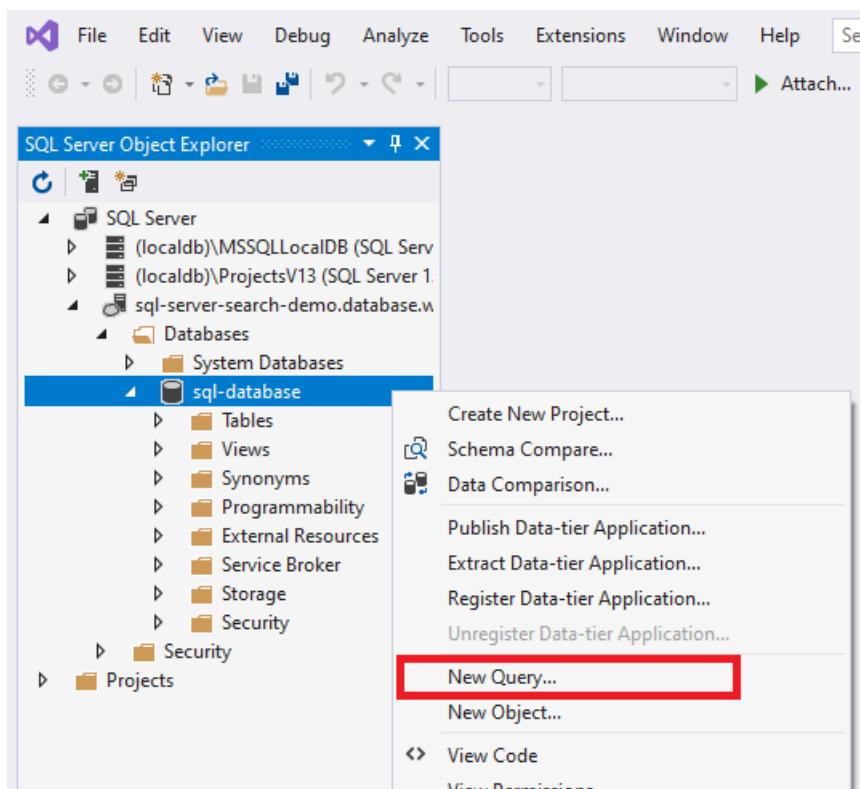
2. Authenticate with your Azure AD account

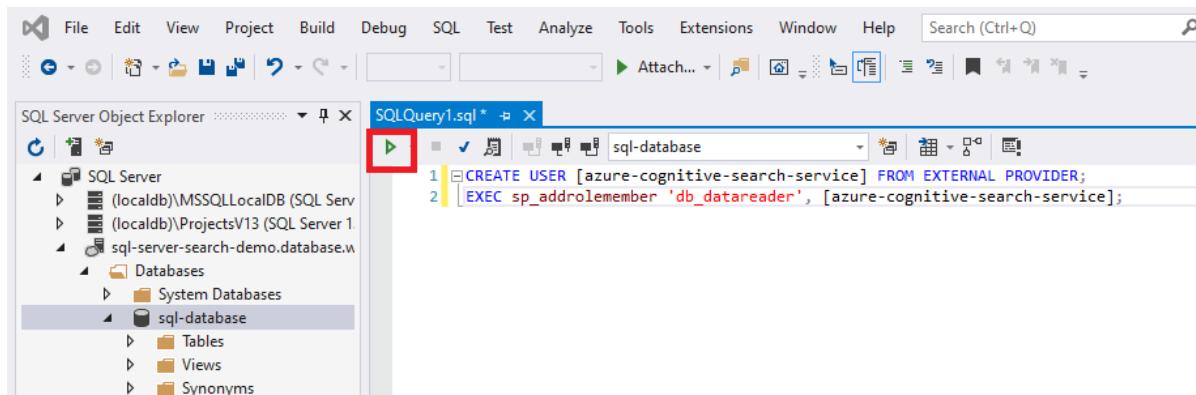


3. Execute the following commands:

Include the brackets around your search service name.

```
CREATE USER [your search service name here] FROM EXTERNAL PROVIDER;
EXEC sp_addrolemember 'db_datareader', [your search service name here];
```





NOTE

If the search service identity from step 1 is changed after completing this step, then you must remove the role membership and remove the user in the SQL database, then add the permissions again by completing step 3 again. Removing the role membership and user can be accomplished by running the following commands:

```
sp_droprolemember 'db_datareader', [your search service name];
DROP USER IF EXISTS [your search service name];
```

4 - Add a role assignment

In this step you will give your Azure Cognitive Search service permission to read data from your SQL Server.

1. In the Azure portal navigate to your Azure SQL Server page.
2. Select Access control (IAM)
3. Select Add then Add role assignment

4. Select the appropriate Reader role.
5. Leave Assign access to as Azure AD user, group or service principal
6. Search for your search service, select it, then select Save

5 - Create the data source

The [REST API](#), Azure portal, and the [.NET SDK](#) support the managed identity connection string. Below is an example of how to create a data source to index data from an Azure SQL Database using the [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.

When creating a data source using the [REST API](#), the data source must have the following required properties:

- **name** is the unique name of the data source within your search service.
- **type** is `azuresql`
- **credentials**
 - When using a managed identity to authenticate, the **credentials** format is different than when not using a managed identity. Here you will 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.
 - Managed identity connection string format:
 - *Initial Catalog/Database=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;*
- **container** specifies the name of the table or view that you would like to index.

Example of how to create an Azure SQL data source object using the [REST API](#):

```
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;ResourceId=/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/azure-sql-resource-group/providers/Microsoft.Sql/servers/sql-server-search-demo;Connection Timeout=30;" },
  "container" : { "name" : "my-table" }
}
```

6 - Create the index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's how to create an index 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 }
    ]
}
```

For more on creating indexes, see [Create Index](#)

7 - 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.

Example indexer definition for an Azure SQL indexer:

```
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",
    "schedule" : { "interval" : "PT2H" }
}
```

This indexer will run every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out [Create Indexer](#).

For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

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](#).

See also

Learn more about the Azure SQL indexer:

- [Azure SQL indexer](#)

Search over Azure Blob storage content

10/4/2020 • 5 minutes to read • [Edit Online](#)

Searching across the variety of content types stored in Azure Blob storage can be a difficult problem to solve. 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.

NOTE

Already familiar with the workflow and composition? [How to configure a blob indexer](#) is your next step.

What it means to add full text search to blob data

Azure Cognitive Search is a 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 blob indexing](#) 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 re-run 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.

Required resources

You need both Azure Cognitive Search and Azure Blob storage. Within Blob storage, you need a container that provides source content.

You can start directly in your Storage account portal page. In the left navigation page, under **Blob service** click **Add Azure Cognitive Search** to create a new service or select an existing one.

Once you add Azure Cognitive Search to your storage account, you can follow the standard process to index blob data. We recommend the **Import data** wizard in Azure Cognitive Search for an easy initial introduction, or call the REST APIs using a tool like Postman. This tutorial walks you through the steps of calling the REST API in Postman: [Index and search semi-structured data \(JSON blobs\) in Azure Cognitive Search](#).

Use a Blob indexer

An *indexer* is a data-source-aware subservice in Cognitive Search, equipped with internal logic for sampling data,

reading metadata data, retrieving data, and serializing data from native formats into JSON documents for subsequent import.

Blobs in Azure Storage are indexed using the [Azure Cognitive Search Blob storage indexer](#). You can invoke this indexer by using 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 does the "document cracking", 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 to text content only.

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 [Azure Cognitive Search Blob storage indexer](#).

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:

- PDF
- 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
- HTML
- XML
- ZIP
- GZ
- EPUB
- EML
- RTF
- Plain text files (see also [Indexing plain text](#))
- JSON (see [Indexing JSON blobs](#))
- CSV (see [Indexing CSV blobs](#))

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](#).

Indexing JSON blobs

Indexers can be configured to extract structured content found in blobs that contain JSON. An indexer can read JSON blobs and parse the structured content into the appropriate fields of a search document. Indexers can also

take blobs that contain an array of JSON objects and map each element to a separate search document. You can set a parsing mode to affect the type of JSON object created by the indexer.

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. You can use [simple query syntax](#), [full query syntax](#), and [filter expression syntax](#) in Search explorer.

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: [Create your first application in Azure Cognitive Search](#).

Next steps

- [Upload, download, and list blobs with the Azure portal \(Azure Blob storage\)](#)
- [Set up a blob indexer \(Azure Cognitive Search\)](#)

Use AI to understand Blob storage data

10/4/2020 • 8 minutes to read • [Edit Online](#)

Data in Azure Blob storage is often a variety of unstructured content such as images, long text, PDFs, and Office documents. By using the AI capabilities in Azure Cognitive Search, you can understand and extract valuable information from blobs in a variety of ways. Examples of applying AI to blob content include:

- Extract text from images using optical character recognition (OCR)
- Produce a scene description or tags from a photo
- Detect language and translate text into different languages
- Infer structure through entity recognition by finding references to people, dates, places, or organizations

While you might need just one of these AI capabilities, it's common to combine multiple of them into the same pipeline (for example, extracting text from a scanned image and then finding all the dates and places referenced in it). It's also common to include custom AI or machine learning processing in the form of leading-edge external packages or in-house models tailored to your data and your requirements.

AI enrichment creates new information, captured as text, stored in fields. Post-enrichment, you can access this information from a search index through full text search, or send enriched documents back to Azure storage to power new application experiences that include exploring data for discovery or analytics scenarios.

In this article, we view AI enrichment through a wide lens so that you can quickly grasp the entire process, from transforming raw data in blobs, to queryable information in either a search index or a knowledge store.

What it means to "enrich" blob data with AI

AI enrichment is part of the indexing architecture of Azure Cognitive Search that integrates built-in AI from Microsoft or custom AI that you provide. It helps you implement end-to-end scenarios where you need to process blobs (both existing ones and new ones as they come in or are updated), crack open all file formats to extract images and text, extract the desired information using various AI capabilities, and index them in a search index for fast search, retrieval and exploration.

Inputs are your blobs, in a single container, in Azure Blob storage. Blobs can be almost any kind of text or image data.

Output is always a search index, used for fast text search, retrieval, and exploration in client applications. Additionally, output can also be a [knowledge store](#) that projects enriched documents into Azure blobs or Azure tables for downstream analysis in tools like Power BI or in data science workloads.

In between is the pipeline architecture itself. The pipeline is based on the *indexer* feature, to which you can assign a *skillset*, which is composed of one or more *skills* providing the AI. The purpose of the pipeline is to produce *enriched documents* that enter as raw content but pick up additional structure, context, and information while moving through the pipeline. Enriched documents are consumed during indexing to create inverted indexes and other structures used in full text search or exploration and analytics.

Required resources

You need Azure Blob storage, Azure Cognitive Search, and a third service or mechanism that provides the AI:

- For built-in AI, Cognitive Search integrates with Azure Cognitive Services vision and natural language processing APIs. You can [attach a Cognitive Services resource](#) to add Optical Character Recognition (OCR), image analysis, or natural language processing (language detection, text translation, entity recognition, key

phrase extraction).

- For custom AI using Azure resources, you can define a custom skill that wraps the external function or model you want to use. [Custom skills](#) can use code provided by Azure Functions, Azure Machine Learning, Azure Form Recognizer, or another resource that is reachable over HTTPS.
- For custom non-Azure AI, your model or module needs to be accessible to an indexer over HTTP.

If you don't have all of the services readily available, start directly in your Storage account portal page. In the left navigation page, under **Blob service** click **Add Azure Cognitive Search** to create a new service or select an existing one.

Once you add Azure Cognitive Search to your storage account, you can follow the standard process to enrich data in any Azure data source. We recommend the **Import data** wizard in Azure Cognitive Search for an easy initial introduction to AI enrichment. You can attach a Cognitive Services resource during the workflow. This quickstart walks you through the steps: [Create an AI enrichment pipeline in the portal](#).

The following sections take a closer look at components and workflow.

Use a Blob indexer

AI enrichment is an add-on to an indexing pipeline, and in Azure Cognitive Search, those pipelines are built on top of an *indexer*. An indexer is a data-source-aware subservice equipped with internal logic for sampling data, reading metadata data, retrieving data, and serializing data from native formats into JSON documents for subsequent import. Indexers are often used by themselves for import, separate from AI, but if you want to build an AI enrichment pipeline, you will need an indexer and a skillset to go with it. This section highlights the indexer; the next section focuses on skillsets.

Blobs in Azure Storage are indexed using the [blob indexer](#). You can invoke this indexer by using the **Import data** wizard, a REST API, or an SDK. A blob indexer is invoked when the data source used by the indexer is an Azure Blob container. You can index a subset of 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 does the "document cracking", 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, image, and other content types are detected. Document cracking with text extraction is no charge. Document cracking with image extraction is charged at rates you can find on the [pricing page](#).

Although all documents will be cracked, enrichment only occurs if you explicitly provide the skills to do so. For example, if your pipeline consists exclusively of image analysis, text in your container or documents is ignored.

The blob indexer comes with configuration parameters and supports change tracking if the underlying data provides sufficient information. You can learn more in [How to configure a blob indexer](#).

Add AI components

AI enrichment refers to modules that look for patterns or characteristics, and then performs an operation accordingly. Facial recognition in photos, text descriptions of photos, detecting key phrases in a document, and OCR (or recognizing printed or handwritten text in binary files) are illustrative examples.

In Azure Cognitive Search, *skills* are the individual components of AI processing that you can use standalone, or in combination with other skills.

- Built-in skills are backed by Cognitive Services, with image analysis based on Computer Vision, and natural language processing based on Text Analytics. For the complete list, see [Built-in skills for content enrichment](#).
- Custom skills are custom code, wrapped in an [interface definition](#) that allows for integration into the pipeline. In customer solutions, it's common practice to use both, with custom skills providing open-source,

third-party, or first-party AI modules.

A *skillset* is the collection of skills used in a pipeline, and it's invoked after the document cracking phase makes content available. An indexer can consume exactly one skillset, but that skillset exists independently of an indexer so that you can reuse it in other scenarios.

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](#).

Built-in skills backed by Cognitive Services require an [attached Cognitive Services](#) all-in-one subscription key that gives you access to the resource. An all-in-one key gives you image analysis, language detection, text translation, and text analytics. Other built-in skills are features of Azure Cognitive Search and require no additional service or key. Text shaper, splitter, and merger are examples of helper skills that are sometimes necessary when designing the pipeline.

If you use only custom skills and built-in utility skills, there is no dependency or costs related to Cognitive Services.

Consume AI-enriched output in downstream solutions

The output of AI enrichment is either a search index on Azure Cognitive Search, or a [knowledge store](#) in Azure Storage.

In Azure Cognitive Search, a search index is used for interactive exploration using free text and filtered queries in a client app. Enriched documents created through AI are formatted in JSON and indexed in the same way all documents are indexed in Azure Cognitive Search, leveraging all of the benefits an indexer provides. For example, during indexing, the blob indexer refers to configuration parameters and settings to utilize any field mappings or change detection logic. Such settings are fully available to regular indexing and AI enriched workloads. Post-indexing, when content is stored on Azure Cognitive Search, you can build rich queries and filter expressions to understand your content.

In Azure Storage, a knowledge store has two manifestations: a blob container, or tables in Table storage.

- A blob container captures enriched documents in their entirety, which is useful if you want to feed into other processes.
- In contrast, Table storage can accommodate physical projections of enriched documents. You can create slices or layers of enriched documents that include or exclude specific parts. For analysis in Power BI, the tables in Azure Table storage become the data source for further visualization and exploration.

An enriched document at the end of the pipeline differs from its original input version by the presence of additional fields containing new information that was extracted or generated during enrichment. As such, you can work with a combination of original and created content, regardless of which output structure you use.

Next steps

There's a lot more you can do with AI enrichment to get the most out of your data in Azure Storage, including combining Cognitive Services in different ways, and authoring custom skills for cases where there's no existing Cognitive Service for the scenario. You can learn more by following the links below.

- [Upload, download, and list blobs with the Azure portal \(Azure Blob storage\)](#)
- [Set up a blob indexer \(Azure Cognitive Search\)](#)
- [AI enrichment overview \(Azure Cognitive Search\)](#)
- [Create a skillset \(Azure Cognitive Search\)](#)
- [Map nodes in an annotation tree \(Azure Cognitive Search\)](#)

How to configure a blob indexer in Azure Cognitive Search

10/4/2020 • 16 minutes to read • [Edit Online](#)

This article shows you how to use Azure Cognitive Search to index text-based documents (such as PDFs, Microsoft Office documents, and several other common formats) stored in Azure Blob storage. First, it explains the basics of setting up and configuring a blob indexer. Then, it offers a deeper exploration of behaviors and scenarios you are likely to encounter.

Supported formats

The blob indexer can extract text from the following document formats:

- PDF
- 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
- HTML
- XML
- ZIP
- GZ
- EPUB
- EML
- RTF
- Plain text files (see also [Indexing plain text](#))
- JSON (see [Indexing JSON blobs](#))
- CSV (see [Indexing CSV blobs](#))

Set up blob indexing

You can set up an Azure Blob Storage indexer using:

- [Azure portal](#)
- Azure Cognitive Search [REST API](#)
- Azure Cognitive Search [.NET SDK](#)

NOTE

Some features (for example, field mappings) are not yet available in the portal, and have to be used programmatically.

Here, we demonstrate the flow using the REST API.

Step 1: Create a data source

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 blob 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 `azureblob`.
- **credentials provide the storage account connection string as the `credentials.connectionString` parameter. See [How to specify credentials](#) below for details.
- **container** specifies a container in your storage account. By default, all blobs within the container are retrievable. If you only want to index blobs in a particular virtual directory, you can specify that directory using the optional **query** parameter.

To create a data source:

```
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" : "DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=<account key>;" },
  "container" : { "name" : "my-container", "query" : "<optional-virtual-directory-name>" }
}
```

For more on the Create Datasource API, see [Create Datasource](#).

How to specify credentials

You can provide the credentials for the blob container in one of these ways:

- **Managed identity connection string:**

```
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 follow the instructions for [Setting up a connection to an Azure Storage account using a managed identity](#).

- **Full access storage account connection string:**

```
DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>
```

You can get the connection string from the Azure portal by navigating to the storage account blade > Settings > Keys (for Classic storage accounts) or Settings > Access keys (for Azure Resource Manager storage accounts).

- **Storage account shared access signature (SAS) connection string:**

```
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=r1
```

The SAS should have the list and read permissions on containers and objects (blobs in this case).

- **Container shared access signature:**

```
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=r1
```

The SAS should have the list and read permissions on the container.

For more information on storage shared access signatures, 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.`.

Step 2: Create an index

The index specifies the fields in a document, attributes, and other constructs that shape the search experience.

Here's how to create an index with a searchable `content` field to store the text extracted from blobs:

```
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 }
    ]
}
```

For more information, see [Create Index \(REST API\)](#).

Step 3: 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:

```
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",
    "schedule" : { "interval" : "PT2H" }
}
```

This indexer will run every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more information, see [Create Indexer \(REST API\)](#). For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

How blobs are indexed

Depending on the [indexer configuration](#), the blob indexer can index storage metadata only (useful when you only care about the metadata and don't need to index the content of blobs), storage and content metadata, or both metadata and textual content. By default, the indexer extracts both metadata and content.

NOTE

By default, blobs with structured content such as JSON or CSV are indexed as a single chunk of text. If you want to index JSON and CSV blobs in a structured way, see [Indexing JSON blobs](#) and [Indexing CSV blobs](#) for more information.

A compound or embedded document (such as a ZIP archive, a Word document with embedded Outlook email containing attachments, or a .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.

- The textual content of the document is extracted into a string field named `content`.

NOTE

Azure Cognitive Search limits how much text it extracts depending 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. A warning is included in the indexer status response for truncated documents.

- User-specified metadata properties present on the blob, if any, are extracted verbatim. Note that this requires a field to be defined in the index with the same name as the metadata key of the blob. For example, if your 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 are extracted into the following fields:
 - `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 should not be stored for later use as it might expire.
- Metadata properties specific to each document format are extracted into the fields listed [here](#).

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.

NOTE

Often, the field names in your existing index will be different from the field names generated during document extraction. You can use **field mappings** to map the property names provided by Azure Cognitive Search to the field names in your search index. You will see an example of field mappings use below.

Defining document keys and field mappings

In Azure Cognitive Search, the document key uniquely identifies a document. Every search index must have exactly one key field of type Edm.String. The key field is required for each document that is being added to the index (it is actually the only required field).

You should carefully consider which extracted field should map to the key field for your index. The candidates are:

- **metadata_storage_name** - this might be a convenient candidate, but note that 1) the names might not be unique, as you may have blobs with the same name in different folders, and 2) the name may contain characters that are invalid in document keys, such as dashes. You can deal with invalid characters by using the `base64Encode` [field mapping function](#) - if you do this, remember to encode document keys when passing them in API calls such as Lookup. (For example, in .NET you can use the [UrlTokenEncode method](#) for that purpose).
- **metadata_storage_path** - using the full path ensures uniqueness, but the path definitely contains `/` characters that are [invalid in a document key](#). As above, you have the option of encoding the keys using the `base64Encode` [function](#).
- A third option is to add a custom metadata property to the blobs. This option does, however, require that your blob upload process adds that metadata property to all blobs. Since the key is a required property, all blobs that don't have that property will fail to be indexed.

IMPORTANT

If there is no explicit mapping for the key field in the index, Azure Cognitive Search automatically uses `metadata_storage_path` as the key and base-64 encodes key values (the second option above).

For this example, let's pick the `metadata_storage_name` field as the document key. Let's also assume your index has a key field named `key` and a field `filesize` for storing the document size. To wire things up as desired, specify the following field mappings when creating or updating your indexer:

```
"fieldMappings" : [
    { "sourceFieldName" : "metadata_storage_name", "targetFieldName" : "key", "mappingFunction" : {
        "name" : "base64Encode" } },
    { "sourceFieldName" : "metadata_storage_size", "targetFieldName" : "fileSize" }
]
```

To bring this all together, here's how you can add field mappings and enable base-64 encoding of keys for an existing indexer:

```

PUT https://[service name].search.windows.net/indexers/blob-indexer?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  "dataSourceName" : "blob-datasource",
  "targetIndexName" : "my-target-index",
  "schedule" : { "interval" : "PT2H" },
  "fieldMappings" : [
    { "sourceFieldName" : "metadata_storage_name", "targetFieldName" : "key", "mappingFunction" : {
      "name" : "base64Encode" } },
    { "sourceFieldName" : "metadata_storage_size", "targetFieldName" : "fileSize" }
  ]
}

```

For more information, see [Field mappings and transformations](#).

What if you need to encode a field to use it as a key, but you also want to search it?

There are times when you need to use an encoded version of a field like metadata_storage_path as the key, but you also need that field to be searchable (without encoding). In order to resolve this issue, you can map it into two fields; one that will be used for the key, and another one that will be used for search purposes. In the example below the *key* field contains the encoded path, while the *path* field is not encoded and will be used as the searchable field in the index.

```

PUT https://[service name].search.windows.net/indexers/blob-indexer?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
  "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" }
  ]
}

```

Index by file type

You can control which blobs are indexed, and which are skipped.

Include blobs having specific file extensions

You can index only the blobs with the file name extensions you specify by using the `indexedFileNameExtensions` indexer configuration parameter. The value is a string containing a comma-separated list of file extensions (with a leading dot). For example, to index only the .PDF and .DOCX blobs, do this:

```

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" : { "indexedFileNameExtensions" : ".pdf,.docx" } }
}

```

Exclude blobs having specific file extensions

You can exclude blobs with specific file name extensions from indexing by using the `excludedFileNameExtensions` configuration parameter. The value is a string containing a comma-separated list of file extensions (with a leading dot). For example, to index all blobs except those with the .PNG and JPEG extensions, do this:

```
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" : { "excludedFileNameExtensions" : ".png,.jpeg" } }
}
```

If both `indexedFileNameExtensions` and `excludedFileNameExtensions` parameters are present, Azure Cognitive Search first looks at `indexedFileNameExtensions`, then at `excludedFileNameExtensions`. This means that if the same file extension is present in both lists, it will be excluded from indexing.

Index parts of a blob

You can control which parts of the blobs are indexed using the `dataToExtract` configuration parameter. It can take the following values:

- `storageMetadata` - specifies that only the [standard blob properties and user-specified metadata](#) are indexed.
- `allMetadata` - specifies that storage metadata and the [content-type specific metadata](#) extracted from the blob content are indexed.
- `contentAndMetadata` - specifies that all metadata and textual content extracted from the blob are indexed. This is the default value.

For example, to index only the storage metadata, use:

```
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" : "storageMetadata" } }
}
```

Using blob metadata to control how blobs are indexed

The configuration parameters described above apply to all blobs. Sometimes, you may want to control how *individual blobs* are indexed. You can do this by adding the following blob metadata properties and values:

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.

PROPERTY NAME	PROPERTY VALUE	EXPLANATION
AzureSearch_SkipContent	"true"	This is equivalent of <div style="border: 1px solid #ccc; padding: 2px; display: inline-block;">"dataToExtract" : "allMetadata"</div> setting described above scoped to a particular blob.

Index from multiple sources

You may want to "assemble" documents from multiple sources in your index. For example, you may want to merge text from blobs with other metadata stored in Cosmos DB. You can even use the push indexing API together with various indexers to build up search documents from multiple parts.

For this to work, all indexers and other components need to agree on the document key. For additional details on this topic, refer to [Index multiple Azure data sources](#) or this blog post, [Combine documents with other data in Azure Cognitive Search](#).

Index large datasets

Indexing blobs can be a time-consuming process. In cases where you have millions of blobs to index, you can speed up indexing by partitioning your data and using multiple indexers to process the data in parallel. Here's how you can set this up:

- Partition your data into multiple blob containers or virtual folders
- Set up several Azure Cognitive Search data sources, one per container or folder. To point to a blob folder, use the `query` parameter:

```
{
  "name" : "blob-datasource",
  "type" : "azureblob",
  "credentials" : { "connectionString" : "<your storage connection string>" },
  "container" : { "name" : "my-container", "query" : "my-folder" }
}
```

- Create a corresponding indexer for each data source. All the indexers can point to the same target search index.
- One search unit in your service can run one indexer at any given time. Creating multiple indexers as described above is only useful if they actually run in parallel. To run multiple indexers in parallel, scale out your search service by creating an appropriate number of partitions and replicas. For example, if your search service has 6 search units (for example, 2 partitions x 3 replicas), then 6 indexers can run simultaneously, resulting in a six-fold increase in the indexing throughput. To learn more about scaling and capacity planning, see [Adjust the capacity of an Azure Cognitive Search service](#).

Handle errors

By default, the blob indexer stops as soon as it encounters a blob with an unsupported content type (for example, an image). You can of course use the `excludedFileNameExtensions` parameter to skip certain content types. However, you may need to index blobs 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`:

```

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 } }
}

```

For some blobs, 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:

```
"parameters" : { "configuration" : { "failOnUnprocessableDocument" : false } }
```

Azure Cognitive Search limits the size of blobs that are indexed. These limits are documented in [Service Limits in Azure Cognitive Search](#). Oversized blobs are treated as errors by default. However, you can still index storage metadata of oversized blobs if you set `indexStorageMetadataOnlyForOversizedDocuments` configuration parameter to true:

```
"parameters" : { "configuration" : { "indexStorageMetadataOnlyForOversizedDocuments" : true } }
```

You can also continue indexing if errors happen at any point of processing, either while parsing blobs 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:

```

{
    ...
    ... other parts of indexer definition
    "parameters" : { "maxFailedItems" : 10, "maxFailedItemsPerBatch" : 10 }
}

```

Content type-specific metadata properties

The following table summarizes processing done for each document format, and describes the metadata properties extracted by Azure Cognitive Search.

DOCUMENT FORMAT / CONTENT TYPE	EXTRACTED METADATA	PROCESSING DETAILS
HTML (text/html)	<code>metadata_content_encoding</code> <code>metadata_content_type</code> <code>metadata_language</code> <code>metadata_description</code> <code>metadata_keywords</code> <code>metadata_title</code>	Strip HTML markup and extract text
PDF (application/pdf)	<code>metadata_content_type</code> <code>metadata_language</code> <code>metadata_author</code> <code>metadata_title</code>	Extract text, including embedded documents (excluding images)

DOCUMENT FORMAT / CONTENT TYPE	EXTRACTED METADATA	PROCESSING DETAILS
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
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
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
WORD 2003 XML (application/vnd.ms-wordml)	metadata_content_type metadata_author metadata_creation_date	Strip XML markup and extract text
XLSX (application/vnd.openxmlformats-officedocument.spreadsheetml.sheet)	metadata_content_type metadata_author metadata_creation_date metadata_last_modified	Extract text, including embedded documents
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

DOCUMENT FORMAT / CONTENT TYPE	EXTRACTED METADATA	PROCESSING DETAILS
PPTX (application/vnd.openxmlformats-officedocument.presentationml.presentation)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_slide_count</code> <code>metadata_title</code>	Extract text, including embedded documents
PPT (application/vnd.ms-powerpoint)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_slide_count</code> <code>metadata_title</code>	Extract text, including embedded documents
PPTM (application/vnd.ms-powerpoint.presentation.macroenabled.12)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_slide_count</code> <code>metadata_title</code>	Extract text, including embedded documents
MSG (application/vnd.ms-outlook)	<code>metadata_content_type</code> <code>metadata_message_from</code> <code>metadata_message_from_email</code> <code>metadata_message_to</code> <code>metadata_message_to_email</code> <code>metadata_message_cc</code> <code>metadata_message_cc_email</code> <code>metadata_message_bcc</code> <code>metadata_message_bcc_email</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_subject</code>	Extract text, including text extracted from attachments. <code>metadata_message_to_email</code> , <code>metadata_message_cc_email</code> and <code>metadata_message_bcc_email</code> are string collections, the rest of the fields are strings.
ODT (application/vnd.oasis.opendocument.text)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_character_count</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>metadata_page_count</code> <code>metadata_word_count</code>	Extract text, including embedded documents
ODS (application/vnd.oasis.opendocument.spreadsheet)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code>	Extract text, including embedded documents
ODP (application/vnd.oasis.opendocument.presentation)	<code>metadata_content_type</code> <code>metadata_author</code> <code>metadata_creation_date</code> <code>metadata_last_modified</code> <code>title</code>	Extract text, including embedded documents

DOCUMENT FORMAT / CONTENT TYPE	EXTRACTED METADATA	PROCESSING DETAILS
ZIP (application/zip)	metadata_content_type	Extract text from all documents in the archive
GZ (application/gzip)	metadata_content_type	Extract text from all documents in the archive
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
XML (application/xml)	metadata_content_type metadata_content_encoding	Strip XML 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
EML (message/rfc822)	metadata_content_type metadata_message_from metadata_message_to metadata_message_cc metadata_creation_date metadata_subject	Extract text, including attachments
RTF (application/rtf)	metadata_content_type metadata_author metadata_character_count metadata_creation_date metadata_page_count metadata_word_count	Extract text
Plain text (text/plain)	metadata_content_type metadata_content_encoding	Extract text

See also

- [Indexers in Azure Cognitive Search](#)
- [Understand blobs using AI](#)
- [Blob indexing overview](#)

How to set up change and deletion detection for blobs in Azure Cognitive Search indexing

10/4/2020 • 4 minutes to read • [Edit Online](#)

After an initial search index is created, you might want to configure subsequent indexer jobs to pick up just those documents that have been created or deleted since the initial run. For search content that originates from Azure Blob storage, change detection occurs automatically when you use a schedule to trigger indexing. By default, the service reindexes only the changed blobs, as determined by the blob's `LastModified` timestamp. In contrast with other data sources supported by search indexers, blobs always have a timestamp, which eliminates the need to set up a change detection policy manually.

Although change detection is a given, deletion detection is not. If you want to detect deleted documents, make sure to use a "soft delete" approach. If you delete the blobs outright, corresponding documents will not be removed from the search index.

There are two ways to implement the soft delete approach. Both are described below.

Native blob soft delete (preview)

IMPORTANT

Support for native blob soft delete is in preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). The [REST API version 2020-06-30-Preview](#) provides this feature. There is currently no portal or .NET SDK support.

NOTE

When using the native blob soft delete policy the document keys for the documents in your index must either be a blob property or blob metadata.

In this method you will use the [native blob soft delete](#) feature offered by Azure Blob storage. If native blob soft delete is enabled on your storage account, your data source has a native soft delete policy set, and the indexer finds a blob that has been transitioned to a soft deleted state, the indexer will remove that document from the index. The native blob soft delete policy is not supported when indexing blobs from Azure Data Lake Storage Gen2.

Use the following steps:

1. Enable [native soft delete for Azure Blob storage](#). We recommend setting the retention policy to a value that's much higher than your indexer interval schedule. This way 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.
2. Configure a native blob soft deletion detection policy on the data source. An example is shown below. Since this feature is in preview, you must use the preview REST API.
3. Run the indexer or set the indexer to run on a schedule. When the indexer runs and processes the blob the document will be removed from the index.

```
PUT https://[service name].search.windows.net/datasources/blob-datasource?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
{
  "name" : "blob-datasource",
  "type" : "azureblob",
  "credentials" : { "connectionString" : "<your storage connection string>" },
  "container" : { "name" : "my-container", "query" : null },
  "dataDeletionDetectionPolicy" : {
    "@odata.type" :"#Microsoft.Azure.Search.NativeBlobSoftDeleteDetectionPolicy"
  }
}
```

Reindexing un-deleted blobs (using native soft delete policies)

If you delete a blob from Azure Blob storage with native soft delete enabled on your storage account, the blob will transition to a soft deleted state, giving you the option to un-delete that blob within the retention period. If you reverse a deletion after the indexer processed it, the indexer will not always index the restored blob. This is because the indexer determines which blobs to index based on the blob's `LastModified` timestamp. When a soft deleted blob is un-deleted, its `LastModified` timestamp does not get updated, so if the indexer has already processed blobs with more recent `LastModified` timestamps, it won't reindex the un-deleted blob.

To make sure that an un-deleted blob is reindexed, you will 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 that it needs to reindex this blob.

Soft delete using custom metadata

In this method you will use a blob's metadata to indicate when a document should be removed from the search index. This method requires two separate actions, deleting the search document from the index, followed by blob deletion in Azure Storage.

Use the following steps:

1. Add a custom metadata key-value pair to the blob to indicate to Azure Cognitive Search that it is logically deleted.
2. Configure a soft deletion column detection policy on the data source. An example is shown below.
3. Once the indexer has processed the blob and deleted the document from the index, you can delete the blob in Azure Blob storage.

For example, the following policy considers a blob to be deleted if it has a metadata property `IsDeleted` with the value `true`:

```
PUT https://[service name].search.windows.net/datasources/blob-datasource?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "blob-datasource",
    "type" : "azureblob",
    "credentials" : { "connectionString" : "<your storage connection string>" },
    "container" : { "name" : "my-container", "query" : null },
    "dataDeletionDetectionPolicy" : {
        "@odata.type" :"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",
        "softDeleteColumnName" : "IsDeleted",
        "softDeleteMarkerValue" : "true"
    }
}
```

Reindexing un-deleted blobs (using custom metadata)

After an indexer processes a deleted blob and removes the corresponding search document from the index, it won't revisit that blob if you restore it later if the blob's `LastModified` timestamp is older than the last indexer run.

If you would like to reindex that document, change the `"softDeleteMarkerValue" : "false"` for that blob and rerun the indexer.

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, provide your input on [UserVoice](#). If you need help using the existing feature, post your question on [Stack Overflow](#).

Next steps

- [Indexers in Azure Cognitive Search](#)
- [How to configure a blob indexer](#)
- [Blob indexing overview](#)

Indexing blobs to produce multiple search documents

10/4/2020 • 3 minutes to read • [Edit Online](#)

By default, a blob indexer will treat the contents of a blob as a single search document. Certain **parsingMode** values support scenarios where an individual blob can result in multiple search documents. The different types of **parsingMode** that allow an indexer to extract more than one search document from a blob are:

- `delimitedText`
- `jsonArray`
- `jsonLines`

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 is no explicit mapping for the key field in the index Azure Cognitive Search automatically maps the `metadata_storage_path` property as the key. This mapping ensures that each blob appears as a distinct search document.

When using any of the parsing modes listed above, 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

```
{ "temperature": 100, "pressure": 100, "timestamp": "2019-02-13T00:00:00Z" }
{ "temperature" : 33, "pressure" : 30, "timestamp": "2019-02-14T00:00:00Z" }
```

Blob2.json

```
{ "temperature": 1, "pressure": 1, "timestamp": "2018-01-12T00:00:00Z" }
{ "temperature" : 120, "pressure" : 3, "timestamp": "2013-05-11T00:00:00Z" }
```

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

```
{  
    "sourceFieldName" : "AzureSearch_DocumentKey",  
    "targetFieldName": "id",  
    "mappingFunction": { "name" : "base64Encode" }  
}
```

This setup will result in the Azure Cognitive Search index containing the following information (base64 encoded ID shortened for brevity)

ID	TEMPERATURE	PRESSURE	TIMESTAMP
aHRO ... YjEuanNvbjsx	100	100	2019-02-13T00:00:00Z
aHRO ... YjEuanNvbjsy	33	30	2019-02-14T00:00:00Z
aHRO ... YjluanNvbjsx	1	1	2018-01-12T00:00:00Z
aHRO ... YjluanNvbjsy	120	3	2013-05-11T00:00:00Z

Custom field mapping for index key field

Assuming the same index definition as the previous example, say your blob container has blobs with the following structure:

Blob1.json

```
recordid, temperature, pressure, timestamp  
1, 100, 100,"2019-02-13T00:00:00Z"  
2, 33, 30,"2019-02-14T00:00:00Z"
```

Blob2.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:

```
{  
    "sourceFieldName" : "recordid",  
    "targetFieldName": "id"  
}
```

However, this mapping will *not* result in 4 documents showing up in the index, because the `recordid` field is not 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 its value for your application's needs.

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, provide your input on [UserVoice](#). If you need help using the existing feature, post your question on [Stack Overflow](#).

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](#)

How to index plain text blobs in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

When using a [blob indexer](#) to extract searchable text for full text search, you can invoke various parsing modes to get better indexing outcomes. By default, the indexer parses delimited text blobs as a single chunk of text. However, if all your blobs contain plain text in the same encoding, you can significantly improve indexing performance by using [text parsing mode](#).

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:

```
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:

```
{
    ... other parts of indexer definition
    "parameters" : { "configuration" : { "parsingMode" : "text", "encoding" : "windows-1252" } }
}
```

Request example

Parsing modes are specified in the indexer definition.

```
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" } }
}
```

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, provide your input on [UserVoice](#). If you need help using the existing feature, post your question on [Stack Overflow](#).

Next steps

- [Indexers in Azure Cognitive Search](#)
- [How to configure a blob indexer](#)
- [Blob indexing overview](#)

How to index CSV blobs using delimitedText parsing mode and Blob indexers in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

By default, [Azure Cognitive Search blob indexer](#) parses delimited text blobs as a single chunk of text. However, with blobs containing CSV data, you often want to treat each line in the blob as a separate document. For example, given the following delimited text, you might want to parse it into two documents, each containing "id", "datePublished", and "tags" fields:

```
id, datePublished, tags
1, 2016-01-12, "azure-search,azure,cloud"
2, 2016-07-07, "cloud,mobile"
```

In this article, you will learn how to parse CSV blobs with an Azure Cognitive Search blob indexer by setting the `delimitedText` parsing mode.

NOTE

Follow the indexer configuration recommendations in [One-to-many indexing](#) to output multiple search documents from one Azure blob.

Setting up CSV indexing

To index CSV blobs, create or update an indexer definition with the `delimitedText` parsing mode on a [Create Indexer](#) request:

```
{
  "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:

```
"parameters" : { "configuration" : { "parsingMode" : "delimitedText", "delimitedTextHeaders" :
"id,datePublished,tags" } }
```

You can customize the delimiter character using the `delimitedTextDelimiter` configuration setting. For example:

```
"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](#).

Request examples

Putting this all together, here are the complete payload examples.

Datasource:

```
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:

```
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" } }
}
```

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, provide your input on [UserVoice](#). If you need help using the existing feature, post your question on [Stack Overflow](#).

How to index JSON blobs using a Blob indexer in Azure Cognitive Search

10/4/2020 • 17 minutes to read • [Edit Online](#)

This article shows you how to configure an Azure Cognitive Search blob [indexer](#) to extract structured content from JSON documents in Azure Blob storage and make it searchable in Azure Cognitive Search. This workflow creates an Azure Cognitive Search index and loads it with existing text extracted from JSON blobs.

You can use the [portal](#), [REST APIs](#), or [.NET SDK](#) to index JSON content. Common to all approaches is that JSON documents are located in a blob container in an Azure Storage account. For guidance on pushing JSON documents from other non-Azure platforms, see [Data import in Azure Cognitive Search](#).

JSON blobs in Azure Blob storage are typically either a single JSON document (parsing mode is `json`) or a collection of JSON entities. For collections, the blob could have an **array** of well-formed JSON elements (parsing mode is `jsonArray`). Blobs could also be composed of multiple individual JSON entities separated by a newline (parsing mode is `jsonLines`). The **parsingMode** parameter on the request determines the output structures.

NOTE

For more information about indexing multiple search documents from a single blob, see [One-to-many indexing](#).

Use the portal

The easiest method for indexing JSON documents is to use a wizard in the [Azure portal](#). By parsing metadata in the Azure blob container, the **Import data** wizard can create a default index, map source fields to target index fields, and load the index in a single operation. Depending on the size and complexity of source data, you could have an operational full text search index in minutes.

We recommend using the same region or location for both Azure Cognitive Search and Azure Storage for lower latency and to avoid bandwidth charges.

1 - Prepare source data

[Sign in to the Azure portal](#) and [create a Blob container](#) to contain your data. The Public Access Level can be set to any of its valid values.

You will need the storage account name, container name, and an access key to retrieve your data in the **Import data** wizard.

2 - Start Import data wizard

In the Overview page of your search service, you can [start the wizard](#) from the command bar.



3 - Set the data source

In the **data source** page, the source must be **Azure Blob Storage**, with the following specifications:

- **Data to extract** should be *Content and metadata*. Choosing this option allows the wizard to infer an index schema and map the fields for import.
- **Parsing mode** should be set to *JSON*, *JSON array* or *JSON lines*.

JSON articulates each blob as a single search document, showing up as an independent item in search results.

JSON array is for blobs that contain well-formed JSON data - the well-formed JSON corresponds to an array of objects, or has a property which is an array of objects and you want each element to be articulated as a standalone, independent search document. If blobs are complex, and you don't choose *JSON array* the entire blob is ingested as a single document.

JSON lines is for blobs composed of multiple JSON entities separated by a new-line, where you want each entity to be articulated as a standalone independent search document. If blobs are complex, and you don't choose *JSON lines* parsing mode, then the entire blob is ingested as a single document.

- **Storage container** must specify your storage account and container, or a connection string that resolves to the container. You can get connection strings on the Blob service portal page.

The screenshot shows the 'New data source' configuration screen. It includes fields for Name (blobdatasource), Data to extract (Content and metadata), Parsing mode (JSON array), Storage container (blobstore/basicdemo), and Blob folder (your/folder/here). The 'Data to extract', 'Parsing mode', and 'Storage container' fields are highlighted with red boxes.

4 - Skip the "Enrich content" page in the wizard

Adding cognitive skills (or enrichment) is not an import requirement. Unless you have a specific need to [add AI enrichment](#) to your indexing pipeline, you should skip this step.

To skip the step, click the blue buttons at the bottom of the page for "Next" and "Skip".

5 - Set index attributes

In the **Index** page, you should see a list of fields with a data type and a series of checkboxes for setting index attributes. The wizard can generate a fields list based on metadata and by sampling the source data.

You can bulk-select attributes by clicking the checkbox at the top of an attribute column. Choose **Retrievable** and **Searchable** for every field that should be returned to a client app and subject to full text search processing. You'll notice that integers are not full text or fuzzy searchable (numbers are evaluated verbatim and are often useful in filters).

Review the description of [index attributes](#) and [language analyzers](#) for more information.

Take a moment to review your selections. Once you run the wizard, physical data structures are created and you won't be able to edit these fields without dropping and recreating all objects.

Index							
			RETRIEVABLE	FILTERABLE	SORTABLE	FACEABLE	SEARCHABLE
content	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
metadata_storage_content_type	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_storage_size	Edm.Int64	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_storage_last_modified	Edm.DateTim...	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_storage_name	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_storage_path	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>				
metadata_content_encoding	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_content_type	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_language	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metadata_title	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Edm.String	<input type="button" value="▼"/>	<input type="checkbox"/>				
OK							

6 - Create indexer

Fully specified, the wizard creates three distinct objects in your search service. A data source object and index object are saved as named resources in your Azure Cognitive Search service. The last step creates an indexer object. Naming the indexer allows it to exist as a standalone resource, which you can schedule and manage independently of the index and data source object, created in the same wizard sequence.

If you are not familiar with indexers, an *indexer* is a resource in Azure Cognitive Search that crawls an external data source for searchable content. The output of the **Import data** wizard is an indexer that crawls your JSON data source, extracts searchable content, and imports it into an index on Azure Cognitive Search.

Create an indexer

* Name

Schedule

Once Hourly Daily Custom

Advanced options >

Click **OK** to run the wizard and create all objects. Indexing commences immediately.

You can monitor data import in the portal pages. Progress notifications indicate indexing status and how many documents are uploaded.

When indexing is complete, you can use [Search explorer](#) to query your index.

NOTE

If you don't see the data you expect, you might need to set more attributes on more fields. Delete the index and indexer you just created, and step through the wizard again, modifying your selections for index attributes in step 5.

Use REST APIs

You can use the REST API to index JSON blobs, following a three-part workflow common to all indexers in Azure Cognitive Search: create a data source, create an index, create an indexer. Data extraction from blob storage occurs when you submit the **Create Indexer** request. After this request is finished, you will have a queryable index.

You can review [REST example code](#) at the end of this section that shows how to create all three objects. This section also contains details about [JSON parsing modes](#), [single blobs](#), [JSON arrays](#), and [nested arrays](#).

For code-based JSON indexing, use [Postman](#) and the REST API to create these objects:

- [index](#)
- [data source](#)
- [indexer](#)

Order of operations requires that you create and call objects in this order. In contrast with the portal workflow, a code approach requires an available index to accept the JSON documents sent through the **Create Indexer** request.

JSON blobs in Azure Blob storage are typically either a single JSON document or a JSON "array". The blob indexer in Azure Cognitive Search can parse either construction, depending on how you set the **parsingMode** parameter on the request.

JSON DOCUMENT	PARSING MODE	DESCRIPTION	AVAILABILITY
One per blob	<code>json</code>	Parses JSON blobs as a single chunk of text. Each JSON blob becomes a single Azure Cognitive Search document.	Generally available in both REST API and .NET SDK .
Multiple per blob	<code>jsonArray</code>	Parses a JSON array in the blob, where each element of the array becomes a separate Azure Cognitive Search document.	Generally available in both REST API and .NET SDK .
Multiple per blob	<code>jsonLines</code>	Parses a blob which contains multiple JSON entities (an "array") separated by a newline, where each entity becomes a separate Azure Cognitive Search document.	Generally available in both REST API and .NET SDK .

1 - Assemble inputs for the request

For each request, you must provide the service name and admin key for Azure Cognitive Search (in the POST header), and the storage account name and key for blob storage. You can use [Postman](#) to send HTTP requests to Azure Cognitive Search.

Copy the following four values into Notepad so that you can paste them into a request:

- Azure Cognitive Search service name
- Azure Cognitive Search admin key
- Azure storage account name
- Azure storage account key

You can find these values in the portal:

1. In the portal pages for Azure Cognitive Search, copy the search service URL from the Overview page.
2. In the left navigation pane, click **Keys** and then copy either the primary or secondary key (they are equivalent).
3. Switch to the portal pages for your storage account. In the left navigation pane, under **Settings**, click **Access Keys**. This page provides both the account name and key. Copy the storage account name and one of the keys to Notepad.

2 - Create a data source

This step provides data source connection information used by the indexer. The data source is a named object in Azure Cognitive Search that persists the connection information. The data source type, `azureblob`, determines which data extraction behaviors are invoked by the indexer.

Substitute valid values for service name, admin key, storage account, and account key placeholders.

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "name" : "my-blob-datasource",
    "type" : "azureblob",
    "credentials" : { "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<account
name>;AccountKey=<account key>" },
    "container" : { "name" : "my-container", "query" : "optional, my-folder" }
}
```

3 - Create a target search index

Indexers are paired with an index schema. If you are using the API (rather than the portal), prepare an index in advance so that you can specify it on the indexer operation.

The index stores searchable content in Azure Cognitive Search. To create an index, provide a schema that specifies the fields in a document, attributes, and other constructs that shape the search experience. If you create an index that has the same field names and data types as the source, the indexer will match the source and destination fields, saving you the work of having to explicitly map the fields.

The following example shows a [Create Index](#) request. The index will have a searchable `content` field to store the text extracted from blobs:

```

POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "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 }
    ]
}

```

4 - Configure and run the indexer

As with an index and a data source, an indexer is also a named object that you create and reuse on an Azure Cognitive Search service. A fully specified request to create an indexer might look as follows:

```

POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "schedule" : { "interval" : "PT2H" },
    "parameters" : { "configuration" : { "parsingMode" : "json" } }
}

```

Indexer configuration is in the body of the request. It requires a data source and an empty target index that already exists in Azure Cognitive Search.

Schedule and parameters are optional. If you omit them, the indexer runs immediately, using `json` as the parsing mode.

This particular indexer does not include field mappings. Within the indexer definition, you can leave out **field mappings** if the properties of the source JSON document match the fields of your target search index.

REST Example

This section is a recap of all the requests used for creating objects. For a discussion of component parts, see the previous sections in this article.

Data source request

All indexers require a data source object that provides connection information to existing data.

```

POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "name" : "my-blob-datasource",
    "type" : "azureblob",
    "credentials" : { "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<account name>;AccountKey=<account key>;" },
    "container" : { "name" : "my-container", "query" : "optional, my-folder" }
}

```

Index request

All indexers require a target index that receives the data. The body of the request defines the index schema, consisting of fields, attributed to support the desired behaviors in a searchable index. This index should be empty when you run the indexer.

```
POST https://[service name].search.windows.net/indexes?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "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 }
    ]
}
```

Indexer request

This request shows a fully-specified indexer. It includes field mappings, which were omitted in previous examples. Recall that "schedule", "parameters", and "fieldMappings" are optional as long as there is an available default. Omitting "schedule" causes the indexer to run immediately. Omitting "parsingMode" causes the index to use the "json" default.

Creating the indexer on Azure Cognitive Search triggers data import. It runs immediately, and thereafter on a schedule if you've provided one.

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key for Azure Cognitive Search]

{
    "name" : "my-json-indexer",
    "dataSourceName" : "my-blob-datasource",
    "targetIndexName" : "my-target-index",
    "schedule" : { "interval" : "PT2H" },
    "parameters" : { "configuration" : { "parsingMode" : "json" } },
    "fieldMappings" : [
        { "sourceFieldName" : "/article/text", "targetFieldName" : "text" },
        { "sourceFieldName" : "/article/datePublished", "targetFieldName" : "date" },
        { "sourceFieldName" : "/article/tags", "targetFieldName" : "tags" }
    ]
}
```

Use .NET SDK

The .NET SDK has full parity with the REST API. We recommend that you review the previous REST API section to learn concepts, workflow, and requirements. You can then refer to following .NET API reference documentation to implement a JSON indexer in managed code.

- [microsoft.azure.search.models.datasource](#)
- [microsoft.azure.search.models.datasourcetype](#)
- [microsoft.azure.search.models.index](#)
- [microsoft.azure.search.models.indexer](#)

Parsing modes

JSON blobs can assume multiple forms. The **parsingMode** parameter on the JSON indexer determines how

JSON blob content is parsed and structured in an Azure Cognitive Search index:

PARSING MODE	DESCRIPTION
json	Index each blob as a single document. This is the default.
jsonArray	Choose this mode if your blobs consist of JSON arrays, and you need each element of the array to become a separate document in Azure Cognitive Search.
jsonLines	Choose this mode if your blobs consist of multiple JSON entities, that are separated by a new line, and you need each entity to become a separate document in Azure Cognitive Search.

You can think of a document as a single item in search results. If you want each element in the array to show up in search results as an independent item, then use the `jsonArray` or `jsonLines` option as appropriate.

Within the indexer definition, you can optionally use [field mappings](#) to choose which properties of the source JSON document are used to populate your target search index. For `jsonArray` parsing mode, if the array exists as a lower-level property, you can set a document root indicating where the array is placed within the blob.

IMPORTANT

When you use `json`, `jsonArray` or `jsonLines` parsing mode, Azure Cognitive Search assumes that all blobs in your data source contain JSON. If you need to support a mix of JSON and non-JSON blobs in the same data source, let us know on [our UserVoice site](#).

Parse single JSON blobs

By default, [Azure Cognitive Search blob indexer](#) parses JSON blobs as a single chunk of text. Often, you want to preserve the structure of your JSON documents. For example, assume you have the following JSON document in Azure Blob storage:

```
{  
  "article" : {  
    "text" : "A hopefully useful article explaining how to parse JSON blobs",  
    "datePublished" : "2016-04-13",  
    "tags" : [ "search", "storage", "howto" ]  
  }  
}
```

The blob indexer parses the JSON document into a single Azure Cognitive Search document. The indexer loads an index by matching "text", "datePublished", and "tags" from the source against identically named and typed target index fields.

As noted, field mappings are not required. 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.

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 Azure Cognitive Search document. For example, given the following JSON blob, you can populate your Azure Cognitive Search index with three separate documents, each with "id" and "text" fields.

```
[  
  { "id" : "1", "text" : "example 1" },  
  { "id" : "2", "text" : "example 2" },  
  { "id" : "3", "text" : "example 3" }  
]
```

For a JSON array, the indexer definition should look similar to the following example. Notice that the `parsingMode` parameter specifies the `jsonArray` parser. Specifying the right parser and having the right data input are the only two array-specific requirements for indexing JSON blobs.

```
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",  
  "schedule" : { "interval" : "PT2H" },  
  "parameters" : { "configuration" : { "parsingMode" : "jsonArray" } }  
}
```

Again, notice that field mappings can be omitted. Assuming an index with identically named "id" and "text" fields, the blob indexer can infer the correct mapping without an explicit field mapping list.

Parse nested 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:

```
{  
  "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:

```
{  
  "name" : "my-json-array-indexer",  
  ... other indexer properties  
  "parameters" : { "configuration" : { "parsingMode" : "jsonArray", "documentRoot" : "/level1/level2" } }  
}
```

Parse blobs separated by newlines

If your blob contains multiple JSON entities separated by a newline, and you want each element to become a separate Azure Cognitive Search document, you can opt for the JSON lines option. For example, given the following blob (where there are three different JSON entities), you can populate your Azure Cognitive Search index with three separate documents, each with "id" and "text" fields.

```
{ "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. Notice that the `parsingMode` parameter specifies the `jsonLines` parser.

```
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",
    "schedule" : { "interval" : "PT2H" },
    "parameters" : { "configuration" : { "parsingMode" : "jsonLines" } }
}
```

Again, notice that field mappings can be omitted, similar to the `jsonArray` parsing mode.

Add field mappings

When source and target fields are not perfectly aligned, you can define a field mapping section in the request body for explicit field-to-field associations.

Currently, Azure Cognitive Search cannot index arbitrary JSON documents directly because it supports only primitive data types, string arrays, and GeoJSON points. However, you can use **field mappings** to pick parts of your JSON document and "lift" them into top-level fields of the search document. To learn about field mappings basics, see [Field mappings in Azure Cognitive Search indexers](#).

Revisiting our example JSON document:

```
{
    "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:

```
"fieldMappings" : [
    { "sourceFieldName" : "/article/text", "targetFieldName" : "text" },
    { "sourceFieldName" : "/article/datePublished", "targetFieldName" : "date" },
    { "sourceFieldName" : "/article/tags", "targetFieldName" : "tags" }
]
```

The source field names in the mappings 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:

```
{ "sourceFieldName" : "/article/tags/0", "targetFieldName" : "firstTag" }
```

NOTE

If a source field name in a field mapping path refers to a property that doesn't exist in JSON, that mapping is skipped without an error. This is done so that we can support documents with a different schema (which is a common use case). Because there is no validation, you need to take care to avoid typos in your field mapping specification.

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, provide your input on [UserVoice](#). If you need help using the existing feature, post your question on [Stack Overflow](#).

See also

- [Indexers in Azure Cognitive Search](#)
- [Indexing Azure Blob Storage with Azure Cognitive Search](#)
- [Indexing CSV blobs with Azure Cognitive Search blob indexer](#)
- [Tutorial: Search semi-structured data from Azure Blob storage](#)

How to index encrypted blobs using blob indexers and skillsets in Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

This article shows how to use [Azure Cognitive Search](#) to index documents that have been previously encrypted within [Azure Blob Storage](#) using [Azure Key Vault](#). Normally, an indexer cannot extract content from encrypted files because it doesn't have access to the encryption key. However, by leveraging the [DecryptBlobFile](#) custom skill followed by the [DocumentExtractionSkill](#), you can provide controlled access to the key to decrypt the files and then have content extracted from them. This unlocks the ability to index these documents while never having to worry about your data being stored unencrypted at rest.

This guide uses Postman and the Search REST APIs to perform the following tasks:

- Start with whole documents (unstructured text) such as PDF, HTML, DOCX, and PPTX in Azure Blob storage that have been encrypted using Azure Key Vault.
- 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

This example assumes that you have already uploaded your files to Azure Blob Storage and have encrypted them in the process. If you need help with getting your files initially uploaded and encrypted, check out [this tutorial](#) for how to do so.

- [Azure Storage](#)
- [Azure Key Vault](#)
- [Azure Function](#)
- [Postman desktop app](#)
- [Create or find an existing search service](#)

NOTE

You can use the free service for this guide. A free search service limits you to three indexes, three indexers, three data sources and three skillsets. This guide creates one of each. Before starting, make sure you have room on your service to accept the new resources.

1 - Create services and collect credentials

Set up 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 set up, 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. Select **the subscription where your Azure Key Vault instance exists** (this guide will not work if you select a different subscription), and either select an existing resource group or create a new one (if you create a new one, you will also need to select a region to deploy to).
3. Select **Review + create**, make sure you agree to the terms, and then select **Create** to deploy the Azure Function.

Home >

Custom deployment

Deploy from a custom template

Basics Review + create

Template

Customized template 2 resources

Edit template Edit parameters

Deployment scope

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * (New)

Resource group * (New) decrypt-blob-files

Parameters

Region * East US

Resource Prefix psdlbf

Storage Account Type Standard_LRS

Review + create < Previous Next : Review + create >

4. Wait for the deployment to finish.
5. Navigate to your Azure Key Vault instance in the portal. [Create an access policy](#) in the Azure Key Vault that grants key access to the custom skill.
 - a. Under **Settings**, select **Access policies**, and then select **Add access policy**

The screenshot shows the 'Access policies' page for a specific Key Vault. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Keys, Secrets, Certificates, and Access policies (which is highlighted with a red box). The main area shows 'Enable Access to:' checkboxes for Azure VM deployment, ARM template deployment, and Azure Disk Encryption. Below that, the 'Permission model' is set to 'Vault access policy' (radio button selected). A red box highlights the '+ Add Access Policy' button. The 'Current Access Policies' table has columns for Name, Email, and Key Permissions.

- b. Under **Configure from template**, select **Azure Data Lake Storage or Azure Storage**.
- c. For the principal, 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**.
- d. Do not select anything for the authorized application option.

The screenshot shows the 'Add access policy' page. It includes fields for 'Configure from template (optional)' (set to 'Azure Data Lake Storage or Azure Storage'), 'Key permissions' (3 selected), 'Secret permissions' (0 selected), 'Certificate permissions' (0 selected), 'Select principal *' (containing 'psdbf-function-app-' and 'Object ID:'), and 'Authorized application' (None selected). A red box highlights the 'Add' button at the bottom.

- e. Be sure to click **Save** on the access policies page before navigating away to actually add the access policy.

-keyvault | Access policies

Key vault

Search (Ctrl+ /)

Save Discard Refresh

Please click the 'Save' button to commit your changes.

Enable Access to:

- Azure Virtual Machines for deployment
- Azure Resource Manager for template deployment
- Azure Disk Encryption for volume encryption

6. Navigate to the **psdbf-function-app** function in the portal, and make a note of the following properties as you will need them later in the guide:

- a. The function URL, which can be found under **Essentials** on the main page for the function.

Home > psdbf-function-app-

psdbf-function-app- Function App

Search (Ctrl+ /)

Browse Refresh Stop Restart Swap Get publish profile Reset publish profile ...

Overview

You have multiple notifications. Click to get more details. →

Essentials

Resource group (change)

Status: Running

Location: West US 2

Subscription (change)

Subscription ID

Operating System: Windows

App Service Plan: WestUS2Plan (Y1: 0)

Properties: See More

Runtime version: 2.0.14494.0

Tags (change): Click here to add tags

- b. The host key code, which can be found by navigating to **App keys**, clicking to show the **default** key, and copying the value.

Home > psdbf-function-app-

psdbf-function-app- Function App

Search (Ctrl+ /)

Refresh

Host keys (all functions)

New host key Show values

Filter host keys

Name	Value	Renew key value
_master	Hidden value. Click to show value	Renew key value
default	Hidden value. Click to show value	Renew key value

System keys

Cognitive Services

AI enrichment and skillset execution are backed by Cognitive Services, including Text Analytics 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. After it processes 20 documents, the indexer will fail unless a Cognitive Services key is attached to the skillset. For larger projects, plan on provisioning Cognitive Services at the pay-as-you-go S0 tier. For more information, see [Attach Cognitive Services](#). Note that a Cognitive Services key is required to run a skillset with more than 20 documents even if none of your selected cognitive skills connect to Cognitive Services (such as with the provided skillset if no skills are added to it).

Azure Cognitive Search

The last component is Azure Cognitive Search, which you can [create in the portal](#). You can use the Free tier to complete this guide.

As with the Azure Function, take a moment to collect the admin key. Further on, when you begin structuring requests, you will need to provide the endpoint and admin api-key used to authenticate each request.

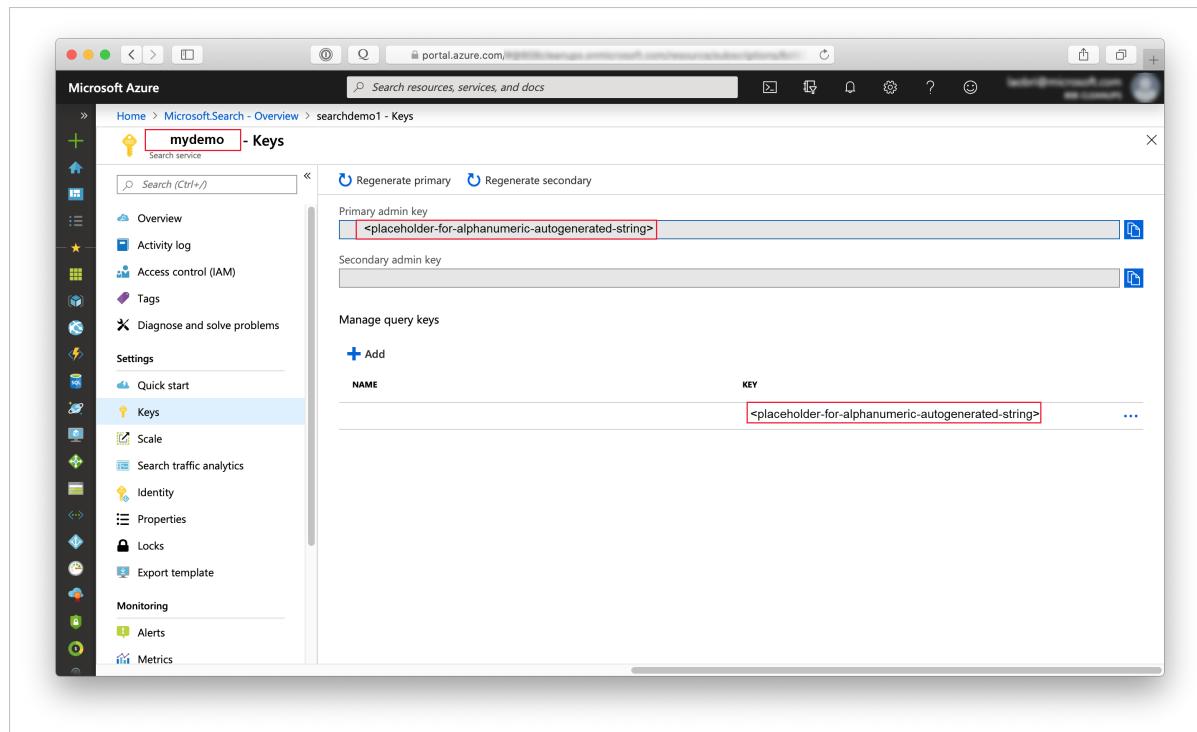
Get an admin api-key and URL for Azure Cognitive Search

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.



The screenshot shows the Microsoft Azure portal interface. The left sidebar has a 'Keys' section selected. The main content area is titled 'mydemo - Keys'. It shows two fields: 'Primary admin key' and 'Secondary admin key', both containing the placeholder text '<placeholder-for-alphanumeric-autogenerated-string>'. Below these fields is a table titled 'Manage query keys' with one entry: a new key named 'NAME' with the same placeholder value in the 'KEY' column.

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

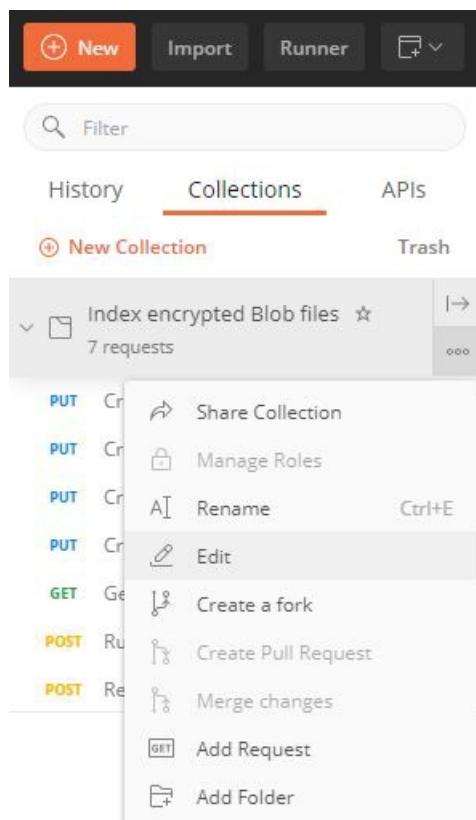
Install and set up Postman.

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.

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.

To get the value for `admin-key`, use the Azure Cognitive Search admin api-key you noted earlier. Set `search-service-name` to the name of the Azure Cognitive Search service you are using. Set `storage-connection-string` by using the value on your storage account's **Access Keys** tab, and set `storage-container-name` to the name of the blob container on that storage account where the encrypted files are stored. Set `function-uri` to the Azure Function URL you noted before, and set `function-code` to the Azure Function host key code you noted before. You can leave the defaults for the other values.

EDIT COLLECTION



Name

Index encrypted Blob files

Description Authorization Pre-request Scripts Tests Variables ●These variables are specific to this collection and its requests. [Learn more about collection variables.](#)

	VARIABLE	INITIAL VALUE <small>i</small>	CURRENT VALUE <small>i</small>	...	Persist All	Reset All
<input checked="" type="checkbox"/>	admin-key	<SEARCH_SERVICE_ADMIN_				
<input checked="" type="checkbox"/>	search-service-name	<SEARCH_SERVICE_NAME>				
<input checked="" type="checkbox"/>	storage-connection-string	<STORAGE_ACCOUNT_CON				
<input checked="" type="checkbox"/>	storage-container-name	<STORAGE_CONTAINER_NA				
<input checked="" type="checkbox"/>	function-uri	<FUNCTION_URI>				
<input checked="" type="checkbox"/>	function-code	<FUNCTION_HOST_CODE>				
<input checked="" type="checkbox"/>	api-version	2020-06-30	2020-06-30			
<input checked="" type="checkbox"/>	datasource-name	encrypted-blobs-ds	encrypted-blobs-ds			
<input checked="" type="checkbox"/>	index-name	encrypted-blobs-idx	encrypted-blobs-idx			
<input checked="" type="checkbox"/>	skillset-name	encrypted-blobs-ss	encrypted-blobs-ss			
<input checked="" type="checkbox"/>	indexer-name	encrypted-blobs-ixr	encrypted-blobs-ixr			
	Add a new variable					

i Use variables to reuse values in different places. Work with the current value of a variable to prevent sharing sensitive values with your team. [Learn more about variable values](#)



Cancel

Update

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 <code>https://{{search-service-name}}.search.windows.net</code> .
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.

VARIABLE	WHERE TO GET IT
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 .
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 the request collection in Postman

When you run this guide, you must issue four HTTP requests:

- **PUT request to create the index:** This index holds the data that Azure Cognitive Search uses and returns.
- **POST request to create the datasource:** This datasource connects your Azure Cognitive Search service to your storage account and therefore 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 reads the data, applies the skillset, and stores the results. You must run this request last.

The [source code](#) contains a Postman collection that has the four requests, as well as some useful follow-up requests. 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.

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 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](#).

How to index tables from Azure Table storage with Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

This article shows how to use Azure Cognitive Search to index data stored in Azure Table storage.

Set up Azure Table storage indexing

You can set up an Azure Table storage indexer by using these resources:

- [Azure portal](#)
- [Azure Cognitive Search REST API](#)
- [Azure Cognitive Search .NET SDK](#)

Here we demonstrate the flow by using the REST API.

Step 1: Create a datasource

A datasource specifies which data to index, the credentials needed to access the data, and the policies that enable Azure Cognitive Search to efficiently identify changes in the data.

For table indexing, the datasource must have the following properties:

- **name** is the unique name of the datasource within your search service.
- **type** must be `azuretable`.
- **credentials** parameter contains the storage account connection string. See the [Specify credentials](#) section for details.
- **container** sets the table name and an optional query.
 - Specify the table name by using the `name` parameter.
 - Optionally, specify a query by using the `query` parameter.

IMPORTANT

Whenever possible, use a filter on PartitionKey for better performance. Any other query does a full table scan, resulting in poor performance for large tables. See the [Performance considerations](#) section.

To create a datasource:

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "table-datasource",
    "type" : "azuretable",
    "credentials" : { "connectionString" : "DefaultEndpointsProtocol=https;AccountName=<account
name>;AccountKey=<account key>;" },
    "container" : { "name" : "my-table", "query" : "PartitionKey eq '123'" }
}
```

For more information on the Create Datasource API, see [Create Datasource](#).

Ways to specify credentials

You can provide the credentials for the table in one of these ways:

- **Managed identity connection string:**

```
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 follow the instructions for [Setting up a connection to an Azure Storage account using a managed identity](#).

- **Full access storage account connection string:**

```
DefaultEndpointsProtocol=https;AccountName=<your storage account>;AccountKey=<your account key>
```

You can get the connection string from the Azure portal by going to the **Storage account blade > Settings > Keys** (for classic storage accounts) or **Settings > Access keys** (for Azure Resource Manager storage accounts).

- **Storage account shared access signature connection string:**

```
TableEndpoint=https://<your account>.table.core.windows.net/;SharedAccessSignature=?sv=2016-05-31&sig=<the signature>&spr=https&se=<the validity end time>&srt=co&ss=t&sp=r1
```

The shared access signature should have the list and read permissions on containers (tables in this case) and objects (table rows).

- **Table shared access signature:**

```
ContainerSharedAccessUri=https://<your storage account>.table.core.windows.net/<table name>?tn=<table name>&sv=2016-05-31&sig=<the signature>&se=<the validity end time>&sp=r
```

The shared access signature should have query (read) permissions on the table.

For more information on storage shared access signatures, see [Using shared access signatures](#).

NOTE

If you use shared access signature credentials, you will need to update the datasource credentials periodically with renewed signatures to prevent their expiration. If shared access signature credentials expire, the indexer fails with an error message similar to "Credentials provided in the connection string are invalid or have expired."

Step 2: Create an index

The index specifies the fields in a document, the attributes, and other constructs that shape the search experience.

To create an index:

```
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": "key", "type": "Edm.String", "key": true, "searchable": false },
        { "name": "SomeColumnInMyTable", "type": "Edm.String", "searchable": true }
    ]
}
```

For more information on creating indexes, see [Create Index](#).

Step 3: Create an indexer

An indexer connects a datasource with a target search index and provides a schedule to automate the data refresh.

After the index and datasource are created, you're ready to create the indexer:

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "table-indexer",
    "dataSourceName" : "table-datasource",
    "targetIndexName" : "my-target-index",
    "schedule" : { "interval" : "PT2H" }
}
```

This indexer runs every two hours. (The schedule interval is set to "PT2H".) To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is five minutes. The schedule is optional; if omitted, an indexer runs only once when it's created. However, you can run an indexer on demand at any time.

For more information on the Create Indexer API, see [Create Indexer](#).

For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

Deal with different field names

Sometimes, the field names in your existing index are different from the property names in your table. You can use field mappings to map the property names from the table to the field names in your search index. To learn more about field mappings, see [Azure Cognitive Search indexer field mappings bridge the differences between datasources and search indexes](#).

Handle document keys

In Azure Cognitive Search, the document key uniquely identifies a document. Every search index must have exactly one key field of type `Edm.String`. The key field is required for each document that is being added to the index. (In fact, it's the only required field.)

Because table rows have a compound key, Azure Cognitive Search generates a synthetic field called `key` that is a concatenation of partition key and row key values. For example, if a row's PartitionKey is `PK1` and RowKey is `RK1`, then the `key` field's value is `PK1RK1`.

NOTE

The `key` value may contain characters that are invalid in document keys, such as dashes. You can deal with invalid characters by using the `base64Encode` [field mapping function](#). If you do this, remember to also use URL-safe Base64 encoding when passing document keys in API calls such as `Lookup`.

Incremental indexing and deletion detection

When you set up a table indexer to run on a schedule, it reindexes only new or updated rows, as determined by a row's `Timestamp` value. You don't have to specify a change detection policy. Incremental indexing is enabled for you automatically.

To indicate that certain documents must be removed from the index, you can use a soft delete strategy. Instead of deleting a row, add a property to indicate that it's deleted, and set up a soft deletion detection policy on the datasource. For example, the following policy considers that a row is deleted if the row has a property `IsDeleted` with the value `"true"`:

```

PUT https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "my-table-datasource",
    "type" : "azuretable",
    "credentials" : { "connectionString" : "<your storage connection string>" },
    "container" : { "name" : "table name", "query" : "<query>" },
    "dataDeletionDetectionPolicy" : { "@odata.type" :
"#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy", "softDeleteColumnName" : "IsDeleted",
"softDeleteMarkerValue" : "true" }
}

```

Performance considerations

By default, Azure Cognitive Search uses the following query filter: `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.

Here are two possible approaches for improving table indexing performance. Both of these approaches rely on using table partitions:

- If your data can naturally be partitioned into several partition ranges, create a datasource 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 datasource 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, you create a new partition every day or week), consider the following approach:
 - Use a query of the form: `(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 complete reindexing, you need to reset the datasource query in addition to resetting the indexer.

Help us make Azure Cognitive Search better

If you have feature requests or ideas for improvements, submit them on our [UserVoice site](#).

How to index Cosmos DB data using an indexer in Azure Cognitive Search

10/4/2020 • 15 minutes to read • [Edit Online](#)

IMPORTANT

SQL API is generally available. MongoDB API, Gremlin API, and Cassandra API support are currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). You can request access to the previews by filling out [this form](#). REST API preview versions provide these features. There is currently limited portal support, and no .NET SDK support.

WARNING

Only Cosmos DB collections with an [indexing policy set to Consistent](#) are supported by Azure Cognitive Search. Indexing collections with a Lazy indexing policy is not recommended and may result in missing data. Collections with indexing disabled are not supported.

This article shows you how to configure an Azure Cosmos DB [indexer](#) to extract content and make it searchable in Azure Cognitive Search. This workflow creates an Azure Cognitive Search index and loads it with existing text extracted from Azure Cosmos DB.

Because terminology can be confusing, it's worth noting that [Azure Cosmos DB indexing](#) and [Azure Cognitive Search indexing](#) are distinct operations, unique to each service. Before you start Azure Cognitive Search indexing, your Azure Cosmos DB database must already exist and contain data.

The Cosmos DB indexer in Azure Cognitive Search can crawl [Azure Cosmos DB items](#) accessed through different protocols.

- For [SQL API](#), which is generally available, you can use the [portal](#), [REST API](#), or [.NET SDK](#) to create the data source and indexer.
- For [MongoDB API \(preview\)](#), you can use either the [portal](#) or the [REST API version 2020-06-30-Preview](#) to create the data source and indexer.
- For [Cassandra API \(preview\)](#) and [Gremlin API \(preview\)](#), you can only use the [REST API version 2020-06-30-Preview](#) to create the data source and indexer.

NOTE

You can cast a vote on User Voice for the [Table API](#) if you'd like to see it supported in Azure Cognitive Search.

Use the portal

NOTE

The portal currently supports the SQL API and MongoDB API (preview).

The easiest method for indexing Azure Cosmos DB items is to use a wizard in the [Azure portal](#). By sampling data and reading metadata on the container, the **Import data** wizard in Azure Cognitive Search can create a default index, map source fields to target index fields, and load the index in a single operation. Depending on the size and complexity of source data, you could have an operational full text search index in minutes.

We recommend using the same region or location for both Azure Cognitive Search and Azure Cosmos DB for lower latency and to avoid bandwidth charges.

1 - Prepare source data

You should have a Cosmos DB account, an Azure Cosmos DB database mapped to the SQL API, MongoDB API (preview), or Gremlin API (preview), and content in the database.

Make sure your Cosmos DB database contains data. The [Import data wizard](#) reads metadata and performs data sampling to infer an index schema, but it also loads data from Cosmos DB. If the data is missing, the wizard stops with this error "Error detecting index schema from data source: Could not build a prototype index because datasource 'emptycollection' returned no data".

2 - Start Import data wizard

You can [start the wizard](#) from the command bar in the Azure Cognitive Search service page, or if you're connecting to Cosmos DB SQL API you can click **Add Azure Cognitive Search** in the **Settings** section of your Cosmos DB account's left navigation pane.



3 - Set the data source

In the [data source](#) page, the source must be **Cosmos DB**, with the following specifications:

- **Name** is the name of the data source object. Once created, you can choose it for other workloads.
- **Cosmos DB account** should be in one of the following formats:

1. The primary or secondary connection string from Cosmos DB with the following format:

```
AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;
```

- For version 3.2 and version 3.6 **MongoDB collections** use the following format for the Cosmos DB account in the Azure portal:

```
AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;ApiKind=MongoDb
```

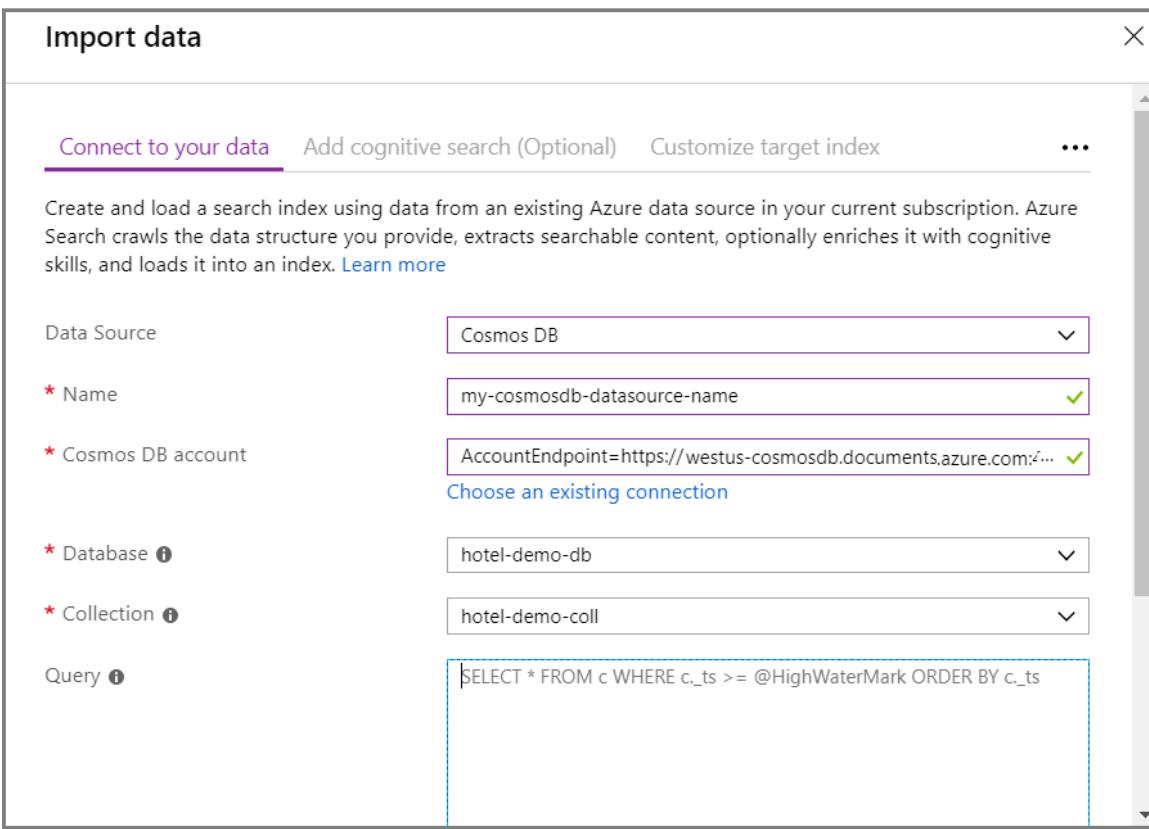
- For **Gremlin graphs and Cassandra tables**, sign up for the [gated indexer preview](#) to get access to the preview and information about how to format the credentials.

2. A managed identity connection string with the following format that does not include an account key:

```
ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>/;(ApiKind=[api-kind]);
```

To use this connection string format, follow the instructions for [Setting up an indexer connection to a Cosmos DB database using a managed identity](#).

- **Database** is an existing database from the account.
- **Collection** is a container of documents. Documents must exist in order for import to succeed.
- **Query** can be blank if you want all documents, otherwise you can input a query that selects a document subset. **Query** is only available for the SQL API.



4 - Skip the "Enrich content" page in the wizard

Adding cognitive skills (or enrichment) is not an import requirement. Unless you have a specific need to [add AI enrichment](#) to your indexing pipeline, you should skip this step.

To skip the step, click the blue buttons at the bottom of the page for "Next" and "Skip".

5 - Set index attributes

In the **Index** page, you should see a list of fields with a data type and a series of checkboxes for setting index attributes. The wizard can generate a fields list based on metadata and by sampling the source data.

You can bulk-select attributes by clicking the checkbox at the top of an attribute column. Choose **Retrievable** and **Searchable** for every field that should be returned to a client app and subject to full text search processing. You'll notice that integers are not full text or fuzzy searchable (numbers are evaluated verbatim and are often useful in filters).

Review the description of [index attributes](#) and [language analyzers](#) for more information.

Take a moment to review your selections. Once you run the wizard, physical data structures are created and you won't be able to edit these fields without dropping and recreating all objects.

Import data

* Key [?](#)
HotellID

Suggerer name Search mode [?](#)

[Delete](#)

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACETABLE	SEARCHABLE	ANALYZER	SUGGESTER
HotellID	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	...
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	...
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	English - Microsoft <input type="button" value="▼"/>	...
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	French - Microsoft <input type="button" value="▼"/>	...
Category	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	...
Tags	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Lucene <input type="button" value="▼"/>	...
ParkingIncluded	Edm.Int64	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...
SmokingAllowed	Edm.Int64	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...
LastRenovationDate	Edm.DateTi...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>			...

[Previous: Add cognitive search \(Optional\)](#) [Next: Create an indexer](#)

6 - Create indexer

Fully specified, the wizard creates three distinct objects in your search service. A data source object and index object are saved as named resources in your Azure Cognitive Search service. The last step creates an indexer object. Naming the indexer allows it to exist as a standalone resource, which you can schedule and manage independently of the index and data source object, created in the same wizard sequence.

If you are not familiar with indexers, an *indexer* is a resource in Azure Cognitive Search that crawls an external data source for searchable content. The output of the **Import data** wizard is an indexer that crawls your Cosmos DB data source, extracts searchable content, and imports it into an index on Azure Cognitive Search.

The following screenshot shows the default indexer configuration. You can switch to **Once** if you want to run the indexer one time. Click **Submit** to run the wizard and create all objects. Indexing commences immediately.

Import data

Connect to your data Add cognitive search (Optional) Customize target index [Create an indexer](#)

Indexer

* Name ✓

Schedule Once Hourly Daily Custom

Change tracking automatically configured with a high watermark policy.

Track deletions

Description

▼ Advanced options

[Previous: Customize target index](#) Submit

You can monitor data import in the portal pages. Progress notifications indicate indexing status and how many documents are uploaded.

When indexing is complete, you can use [Search explorer](#) to query your index.

NOTE

If you don't see the data you expect, you might need to set more attributes on more fields. Delete the index and indexer you just created, and step through the wizard again, modifying your selections for index attributes in step 5.

Use REST APIs

You can use the REST API to index Azure Cosmos DB data, following a three-part workflow common to all indexers in Azure Cognitive Search: create a data source, create an index, create an indexer. Data extraction from Cosmos DB occurs when you submit the Create Indexer request. After this request is finished, you will have a queryable index.

NOTE

For indexing data from Cosmos DB Gremlin API or Cosmos DB Cassandra API you must first request access to the gated previews by filling out [this form](#). Once your request is processed, you will receive instructions for how to use the [REST API version 2020-06-30-Preview](#) to create the data source.

Earlier in this article it is mentioned that [Azure Cosmos DB indexing](#) and [Azure Cognitive Search indexing](#)

indexing are distinct operations. For Cosmos DB indexing, by default all documents are automatically indexed except with the Cassandra API. If you turn off automatic indexing, documents can be accessed only through their self-links or by queries by using the document ID. Azure Cognitive Search indexing requires Cosmos DB automatic indexing to be turned on in the collection that will be indexed by Azure Cognitive Search. When signing up for the Cosmos DB Cassandra API indexer preview, you'll be given instructions on how set up Cosmos DB indexing.

WARNING

Azure Cosmos DB is the next generation of DocumentDB. Previously with API version 2017-11-11 you could use the `documentdb` syntax. This meant that you could specify your data source type as `cosmosdb` or `documentdb`. Starting with API version 2019-05-06 both the Azure Cognitive Search APIs and Portal only support the `cosmosdb` syntax as instructed in this article. This means that the data source type must `cosmosdb` if you would like to connect to a Cosmos DB endpoint.

1 - Assemble inputs for the request

For each request, you must provide the service name and admin key for Azure Cognitive Search (in the POST header), and the storage account name and key for blob storage. You can use [Postman](#) to send HTTP requests to Azure Cognitive Search.

Copy the following four values into Notepad so that you can paste them into a request:

- Azure Cognitive Search service name
- Azure Cognitive Search admin key
- Cosmos DB connection string

You can find these values in the portal:

1. In the portal pages for Azure Cognitive Search, copy the search service URL from the Overview page.
2. In the left navigation pane, click **Keys** and then copy either the primary or secondary key (they are equivalent).
3. Switch to the portal pages for your Cosmos storage account. In the left navigation pane, under **Settings**, click **Keys**. This page provides a URI, two sets of connection strings, and two sets of keys. Copy one of the connection strings to Notepad.

2 - Create a data source

A **data source** specifies the data to index, credentials, and policies for identifying changes in the data (such as modified or deleted documents inside your collection). The data source is defined as an independent resource so that it can be used by multiple indexers.

To create a data source, formulate a POST request:

```

POST https://[service name].search.windows.net/datasources?api-version=2020-06-30
Content-Type: application/json
api-key: [Search service admin key]

{
    "name": "mycosmosdbdatasource",
    "type": "cosmosdb",
    "credentials": {
        "connectionString": "AccountEndpoint=https://myCosmosDbEndpoint.documents.azure.com;AccountKey=myCosmosDbAuthKey;Database=myCosmosDbDatabaseId"
    },
    "container": { "name": "myCollection", "query": null },
    "dataChangeDetectionPolicy": {
        "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
        "highWaterMarkColumnName": "_ts"
    }
}

```

The body of the request contains the data source definition, which should include the following fields:

FIELD	DESCRIPTION
name	Required. Choose any name to represent your data source object.
type	Required. Must be <code>cosmosdb</code> .

FIELD	DESCRIPTION
credentials	<p>Required. Must either follow the Cosmos DB connection string format or a managed identity connection string format.</p> <p>For SQL collections, connection strings can follow either of the below formats:</p> <ul style="list-style-type: none"> • <pre>AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id></pre> • A managed identity connection string with the following format that does not include an account key: <pre>ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>;</pre>. To use this connection string format, follow the instructions for Setting up an indexer connection to a Cosmos DB database using a managed identity. <p>For version 3.2 and version 3.6 MongoDB collections use either of the following formats for the connection string:</p> <ul style="list-style-type: none"> • <pre>AccountEndpoint=https://<Cosmos DB account name>.documents.azure.com;AccountKey=<Cosmos DB auth key>;Database=<Cosmos DB database id>;ApiKind=MongoDb</pre> • A managed identity connection string with the following format that does not include an account key: <pre>ResourceId=/subscriptions/<your subscription ID>/resourceGroups/<your resource group name>/providers/Microsoft.DocumentDB/databaseAccounts/<your cosmos db account name>;ApiKind=MongoDb</pre>. To use this connection string format, follow the instructions for Setting up an indexer connection to a Cosmos DB database using a managed identity. <p>For Gremlin graphs and Cassandra tables, sign up for the gated indexer preview to get access to the preview and information about how to format the credentials.</p> <p>Avoid port numbers in the endpoint url. If you include the port number, Azure Cognitive Search will be unable to index your Azure Cosmos DB database.</p>
container	Contains the following elements: name : Required. Specify the ID of the database collection to be indexed. query : Optional. You can specify a query to flatten an arbitrary JSON document into a flat schema that Azure Cognitive Search can index. For the MongoDB API, Gremlin API, and Cassandra API, queries are not supported.
dataChangeDetectionPolicy	Recommended. See Indexing Changed Documents section.
dataDeletionDetectionPolicy	Optional. See Indexing Deleted Documents section.

Using queries to shape indexed data

You can specify a SQL query to flatten nested properties or arrays, project JSON properties, and filter the data to be indexed.

WARNING

Custom queries are not supported for MongoDB API, Gremlin API, and Cassandra API: `container.query` parameter must be set to null or omitted. If you need to use a custom query, please let us know on [User Voice](#).

Example document:

```
{  
    "userId": 10001,  
    "contact": {  
        "firstName": "andy",  
        "lastName": "hoh"  
    },  
    "company": "microsoft",  
    "tags": ["azure", "cosmosdb", "search"]  
}
```

Filter query:

```
SELECT * FROM c WHERE c.company = "microsoft" and c._ts >= @HighWaterMark ORDER BY c._ts
```

Flattening query:

```
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:

```
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:

```
SELECT c.id, c.userId, tag, c._ts FROM c JOIN tag IN c.tags WHERE c._ts >= @HighWaterMark ORDER BY c._ts
```

3 - Create a target search index

[Create a target Azure Cognitive Search index](#) if you don't have one already. The following example creates an index with an ID and description field:

```

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,
            "sortable": false,
            "facetable": false,
            "suggestable": true
        }
    ]
}

```

Ensure that the schema of your target index is compatible with the schema of the source JSON documents or the output of your custom query projection.

NOTE

For partitioned collections, the default document key is Azure Cosmos DB's `_rid` property, which Azure Cognitive Search automatically renames to `rid` because field names cannot 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.

For MongoDB collections, Azure Cognitive Search automatically renames the `_id` property to `id`.

Mapping between JSON Data Types and Azure Cognitive Search Data Types

JSON DATA TYPE	COMPATIBLE TARGET INDEX 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, for example <code>["a", "b", "c"]</code>	Collection(Edm.String)
Strings that look like dates	Edm.DateTimeOffset, Edm.String
GeoJSON objects, for example <code>{ "type": "Point", "coordinates": [long, lat] }</code>	Edm.GeographyPoint
Other JSON objects	N/A

4 - Configure and run the indexer

Once the index and data source have been created, you're ready to create the indexer:

```
POST https://[service name].search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: [admin key]

{
    "name" : "mycosmosdbindexer",
    "dataSourceName" : "mycosmosdbdatasource",
    "targetIndexName" : "mysearchindex",
    "schedule" : { "interval" : "PT2H" }
}
```

This indexer runs every two hours (schedule interval is set to "PT2H"). To run an indexer every 30 minutes, set the interval to "PT30M". The shortest supported interval is 5 minutes. The schedule is optional - if omitted, an indexer runs only once when it's created. However, you can run an indexer on-demand at any time.

For more details on the Create Indexer API, check out [Create Indexer](#).

For more information about defining indexer schedules, see [How to schedule indexers for Azure Cognitive Search](#).

Use .NET

The generally available .NET SDK has full parity with the generally available REST API. We recommend that you review the previous REST API section to learn concepts, workflow, and requirements. You can then refer to following .NET API reference documentation to implement a JSON indexer in managed code.

- [microsoft.azure.search.models.datasource](#)
- [microsoft.azure.search.models.datasourcetype](#)
- [microsoft.azure.search.models.index](#)
- [microsoft.azure.search.models.indexer](#)

Indexing changed documents

The purpose of a data change detection policy is to efficiently identify changed data items. Currently, the only supported policy is the [HighWaterMarkChangeDetectionPolicy](#) using the `_ts` (timestamp) property provided by Azure Cosmos DB, which is specified as follows:

```
{
    "@odata.type" : "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",
    "highWaterMarkColumnName" : "_ts"
}
```

Using this policy is highly recommended to ensure good indexer performance.

If you are using a custom query, make sure that the `_ts` property is projected by the query.

Incremental progress and custom queries

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 reindex the entire collection from scratch. This is especially important when indexing large collections.

To enable incremental progress when using a custom query, ensure that your 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 using the `assumeOrderByHighWaterMarkColumn` configuration property. To specify this hint, create or update your indexer as follows:

```
{  
    ... 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 as follows:

```
{  
    "@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 are 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:

```
POST https://[service name].search.windows.net/datasources?api-version=2020-06-30  
Content-Type: application/json  
api-key: [Search service admin key]  
  
{  
    "name": "mycosmosdbdatasource",  
    "type": "cosmosdb",  
    "credentials": {  
        "connectionString":  
        "AccountEndpoint=https://myCosmosDbEndpoint.documents.azure.com;AccountKey=myCosmosDbAuthKey;Database=myCosmosDbDatabaseId"  
    },  
    "container": { "name": "myCosmosDbCollectionId" },  
    "dataChangeDetectionPolicy": {  
        "@odata.type": "#Microsoft.Azure.Search.HighWaterMarkChangeDetectionPolicy",  
        "highWaterMarkColumnName": "_ts"  
    },  
    "dataDeletionDetectionPolicy": {  
        "@odata.type": "#Microsoft.Azure.Search.SoftDeleteColumnDeletionDetectionPolicy",  
        "softDeleteColumnName": "isDeleted",  
        "softDeleteMarkerValue": "true"  
    }  
}
```

Next steps

Congratulations! You have learned how to integrate Azure Cosmos DB with Azure Cognitive Search using an indexer.

- To learn more about Azure Cosmos DB, see the [Azure Cosmos DB service page](#).
- To learn more about Azure Cognitive Search, see the [Search service page](#).

Indexing documents in Azure Data Lake Storage Gen2

10/4/2020 • 2 minutes to read • [Edit Online](#)

IMPORTANT

Azure Data Lake Storage Gen2 support is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). You can request access to the previews by filling out [this form](#). The REST API version 2020-06-30-Preview and portal provide this feature. There is currently no .NET SDK support.

When setting up an Azure storage account, you have the option to enable [hierarchical namespace](#). This allows the collection of content in an account to be organized into a hierarchy of directories and nested subdirectories. By enabling hierarchical namespace, you enable [Azure Data Lake Storage Gen2](#).

This article describes how to get started with indexing documents that are in Azure Data Lake Storage Gen2.

Set up Azure Data Lake Storage Gen2 indexer

There are a few steps you'll need to complete to index content from Data Lake Storage Gen2.

Step 1: Sign up for the preview

Sign up for the Data Lake Storage Gen2 indexer preview by filling out [this form](#). You will receive a confirmation email once you have been accepted into the preview.

Step 2: Follow the Azure Blob storage indexing setup steps

Once you've received confirmation that your preview sign-up was successful, you're ready to create the indexing pipeline.

You can index content and metadata from Data Lake Storage Gen2 by using the [REST API version 2020-06-30-Preview](#) or the portal. There is no .NET SDK support at this time.

Indexing content in Data Lake Storage Gen2 is identical to indexing content in Azure Blob storage. So to understand how to set up the Data Lake Storage Gen2 data source, index, and indexer, refer to [How to index documents in Azure Blob Storage with Azure Cognitive Search](#). The Blob storage article also provides information about what document formats are supported, what blob metadata properties are extracted, incremental indexing, and more. This information will be the same for Data Lake Storage Gen2.

Access control

Azure Data Lake Storage Gen2 implements an [access control model](#) that supports both Azure role-based access control (Azure RBAC) and POSIX-like access control lists (ACLs). When indexing content from Data Lake Storage Gen2, Azure Cognitive Search will not extract the RBAC and ACL information from the content. As a result, this information will not be included in your Azure Cognitive Search index.

If maintaining access control on each document in the index is important, it is up to the application developer to implement [security trimming](#).

Change Detection

The Data Lake Storage Gen2 indexer supports change detection. This means that when the indexer runs it only reindexes the changed blobs as determined by the blob's `LastModified` timestamp.

NOTE

Data Lake Storage 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 reindex 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 reindex them during a future run.

Connect to and index Azure SQL content using an Azure Cognitive Search indexer

10/4/2020 • 14 minutes to read • [Edit Online](#)

Before you can query an [Azure Cognitive Search index](#), you must populate it with your data. If the data lives in Azure SQL Database or SQL Managed Instance, an **Azure Cognitive Search indexer for Azure SQL Database** (or **Azure SQL indexer** for short) can automate the indexing process, which means less code to write and less infrastructure to care about.

This article covers the mechanics of using [indexers](#), but also describes features only available with Azure SQL Database or SQL Managed Instance (for example, integrated change tracking).

In addition to Azure SQL Database and SQL Managed Instance, Azure Cognitive Search provides indexers for [Azure Cosmos DB](#), [Azure Blob storage](#), and [Azure table storage](#). To request support for other data sources, provide your feedback on the [Azure Cognitive Search feedback forum](#).

Indexers and data sources

A **data source** specifies which data to index, credentials for data access, and policies that efficiently identify changes in the data (new, modified, or deleted rows). It's defined as an independent resource so that it can be used by multiple indexers.

An **indexer** is a resource that connects a single data source with a targeted search index. An indexer is used in the following ways:

- Perform a one-time copy of the data to populate an index.
- Update an index with changes in the data source on a schedule.
- Run on-demand to update an index as needed.

A single indexer can only consume one table or view, but you can create multiple indexers if you want to populate multiple search indexes. For more information on concepts, see [Indexer Operations: Typical workflow](#).

You can set up and configure an Azure SQL indexer using:

- Import Data wizard in the [Azure portal](#)
- Azure Cognitive Search [.NET SDK](#)
- Azure Cognitive Search [REST API](#)

In this article, we'll use the REST API to create **indexers** and **data sources**.

When to use Azure SQL Indexer

Depending on several factors relating to your data, the use of Azure SQL indexer may or may not be appropriate. If your data fits the following requirements, you can use Azure SQL indexer.

CRITERIA	DETAILS
----------	---------

CRITERIA	DETAILS
Data originates from a single table or view	If the data is scattered across multiple tables, you can create a single view of the data. However, if you use a view, you won't be able to use SQL Server integrated change detection to refresh an index with incremental changes. For more information, see Capturing Changed and Deleted Rows below.
Data types are compatible	Most but not all the SQL types are supported in an Azure Cognitive Search index. For a list, see Mapping data types .
Real-time data synchronization is not required	An indexer can reindex your table at most every five minutes. If your data changes frequently, and the changes need to be reflected in the index within seconds or single minutes, we recommend using the REST API or .NET SDK to push updated rows directly.
Incremental indexing is possible	If you have a large data set and plan to run the indexer on a schedule, Azure Cognitive Search must be able to efficiently identify new, changed, or deleted rows. Non-incremental indexing is only allowed if you're indexing on demand (not on schedule), or indexing fewer than 100,000 rows. For more information, see Capturing Changed and Deleted Rows below.

NOTE

Azure Cognitive Search supports SQL Server authentication only. If you require support for Azure Active Directory Password authentication, please vote for this [UserVoice suggestion](#).

Create an Azure SQL Indexer

1. Create the data source:

```
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" : "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" }
}
```

The connection string can follow either of the below formats:

- You can get the connection string from the [Azure portal](#); use the `ADO.NET connection string` option.
- A managed identity connection string that does not include an account key 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;
```

To use this connection string, follow the instructions for [Setting up an indexer connection to an Azure SQL Database using a managed identity](#).

2. Create the target Azure Cognitive Search index if you don't have one already. You can create an index

using the [portal](#) or the [Create Index API](#). Ensure that the schema of your target index is compatible with the schema of the source table - see [mapping between SQL and Azure Cognitive search data types](#).

3. Create the indexer by giving it a name and referencing the data source and target index:

```
POST https://myservice.search.windows.net/indexers?api-version=2020-06-30
Content-Type: application/json
api-key: admin-key

{
    "name" : "myindexer",
    "dataSourceName" : "myazuresqldatasource",
    "targetIndexName" : "target index name"
}
```

An indexer created in this way doesn't have a schedule. It automatically runs once when it's created. You can run it again at any time using a **run indexer** request:

```
POST https://myservice.search.windows.net/indexers/myindexer/run?api-version=2020-06-30
api-key: admin-key
```

You can customize several aspects of indexer behavior, such as batch size and how many documents can be skipped before an indexer execution fails. For more information, see [Create Indexer API](#).

You may need to allow Azure services to connect to your database. See [Connecting From Azure](#) for instructions on how to do that.

To monitor the indexer status and execution history (number of items indexed, failures, etc.), use an **indexer status** request:

```
GET https://myservice.search.windows.net/indexers/myindexer/status?api-version=2020-06-30
api-key: admin-key
```

The response should look similar to the following:

```
{
    "@odata.context": "https://myservice.search.windows.net/$metadata#Microsoft.Azure.Search.V2015_02_28.IndexerExecutionInfo",
    "status": "running",
    "lastResult": {
        "status": "success",
        "errorMessage": null,
        "startTime": "2015-02-21T00:23:24.957Z",
        "endTime": "2015-02-21T00:36:47.752Z",
        "errors": [],
        "itemsProcessed": 1599501,
        "itemsFailed": 0,
        "initialTrackingState": null,
        "finalTrackingState": null
    },
    "executionHistory": [
        {
            "status": "success",
            "errorMessage": null,
            "startTime": "2015-02-21T00:23:24.957Z",
            "endTime": "2015-02-21T00:36:47.752Z",
            "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 in the response). Additional information about the response can be found in [Get Indexer Status](#)

Run indexers on a schedule

You can also arrange the indexer to run periodically on a schedule. To do this, add the **schedule** property when creating or updating the indexer. The example below shows a PUT request to update the indexer:

```
PUT https://myservice.search.windows.net/indexers/myindexer?api-version=2020-06-30
Content-Type: application/json
api-key: admin-key

{
    "dataSourceName" : "myazuresqldatasource",
    "targetIndexName" : "target index name",
    "schedule" : { "interval" : "PT10M", "startTime" : "2015-01-01T00:00:00Z" }
}
```

The **interval** parameter is required. The interval refers to the time between the start of two consecutive indexer executions. The smallest allowed interval is 5 minutes; the longest is one day. 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 2 hours.

For more information about defining indexer schedules see [How to schedule indexers for Azure Cognitive Search](#).

Capture new, changed, and deleted rows

Azure Cognitive Search uses **incremental indexing** to avoid having to reindex the entire table or view every time an indexer runs. Azure Cognitive Search provides two change detection policies to support incremental indexing.

SQL Integrated Change Tracking Policy

If your SQL database supports [change tracking](#), we recommend using **SQL Integrated Change Tracking Policy**. This is the most efficient policy. In addition, it allows Azure Cognitive Search to identify deleted rows without you having to add an explicit "soft delete" column to your table.

Requirements

- Database version 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.

Usage

To use this policy, create or update your data source like this:

```
{  
    "name" : "myazuresqldatasource",  
    "type" : "azuresql",  
    "credentials" : { "connectionString" : "connection string" },  
    "container" : { "name" : "table or view name" },  
    "dataChangeDetectionPolicy" : {  
        "@odata.type" : "#Microsoft.Azure.Search.SqlIntegratedChangeTrackingPolicy"  
    }  
}
```

When using SQL integrated change tracking policy, do not specify a separate data deletion detection policy - this policy has built-in support for identifying deleted rows. However, for the deletes to be detected "automagically", 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 capturing 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.

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]
```

IMPORTANT

We strongly recommend using the `rowversion` data type for the high water mark column. If any other data type is used, change tracking is not 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.

Usage

To use a high water mark policy, create or update your data source like this:

```
{  
    "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]"  
    }  
}
```

WARNING

If the source table does not 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 using the `convertHighWaterMarkToRowVersion` indexer configuration setting. `convertHighWaterMarkToRowVersion` does two things:

- Use the `rowversion` data type for the high water mark column in the indexer sql query. Using the correct data type improves indexer query performance.
- Subtract 1 from the `rowversion` value before the indexer query runs. Views with 1 to many joins may have rows with duplicate `rowversion` values. Subtracting 1 ensures the indexer query doesn't miss these rows.

To enable this feature, create or update the indexer with the following configuration:

```
{  
    ... other indexer definition properties  
    "parameters" : {  
        "configuration" : { "convertHighWaterMarkToRowVersion" : true } }  
}
```

queryTimeout

If you encounter timeout errors, you can use the `queryTimeout` indexer configuration setting to set the query timeout 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:

```
{  
    ... 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 is not 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 by the time it was interrupted. To disable the `ORDER BY` clause, use the `disableOrderByHighWaterMarkColumn` setting in the indexer definition:

```
{  
    ... 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:

```
{  
    ...  
    "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 – 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.

Mapping between SQL and Azure Cognitive Search data types

SQL DATA TYPE	ALLOWED TARGET INDEX 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 does not support converting decimal types into Edm.Double because this would lose precision

SQL DATA TYPE	ALLOWED TARGET INDEX FIELD TYPES	NOTES
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	
uniqueidentifier	Edm.String	
geography	Edm.GeographyPoint	Only geography instances of type POINT with SRID 4326 (which is the default) are supported
rowversion	N/A	Row-version columns cannot be stored in the search index, but they can be used for change tracking
time, timespan, binary, varbinary, image, xml, geometry, CLR types	N/A	Not supported

Configuration Settings

SQL indexer exposes several configuration settings:

SETTING	DATA TYPE	PURPOSE	DEFAULT VALUE
queryTimeout	string	Sets the timeout for SQL query execution	5 minutes ("00:05:00")
disableOrderByHighWaterMarkColumn	bool	Causes the SQL query used by the high water mark policy to omit the ORDER BY clause. See High Water Mark policy	false

These settings are used in the `parameters.configuration` object in the indexer definition. For example, to set the query timeout to 10 minutes, create or update the indexer with the following configuration:

```
{
    ... other indexer definition properties
    "parameters" : {
        "configuration" : { "queryTimeout" : "00:10:00" } }
}
```

FAQ

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 do not 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 Azure SQL indexer with databases other than SQL Server running in IaaS on Azure?

No. We don't support this scenario, because we haven't tested the indexer with any databases other than SQL Server.

Q: Can I create multiple indexers running on a schedule?

Yes. However, only one indexer can be running on one node at one time. If you need multiple indexers running concurrently, consider scaling up your search service to more than one search unit.

Q: Does running an indexer affect my query workload?

Yes. Indexer runs on one of the nodes in your search service, and that node's resources are shared between indexing and serving query traffic and other API requests. If you run intensive indexing and query workloads and encounter a high rate of 503 errors or increasing response times, consider [scaling up your search service](#).

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 does not 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 is not supported on read-only replicas. Therefore, you must point the indexer to a primary replica if you are using rowversion.

If you attempt to use rowversion on a read-only replica, you will see the following error:

"Using a rowversion column for change tracking is not 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.

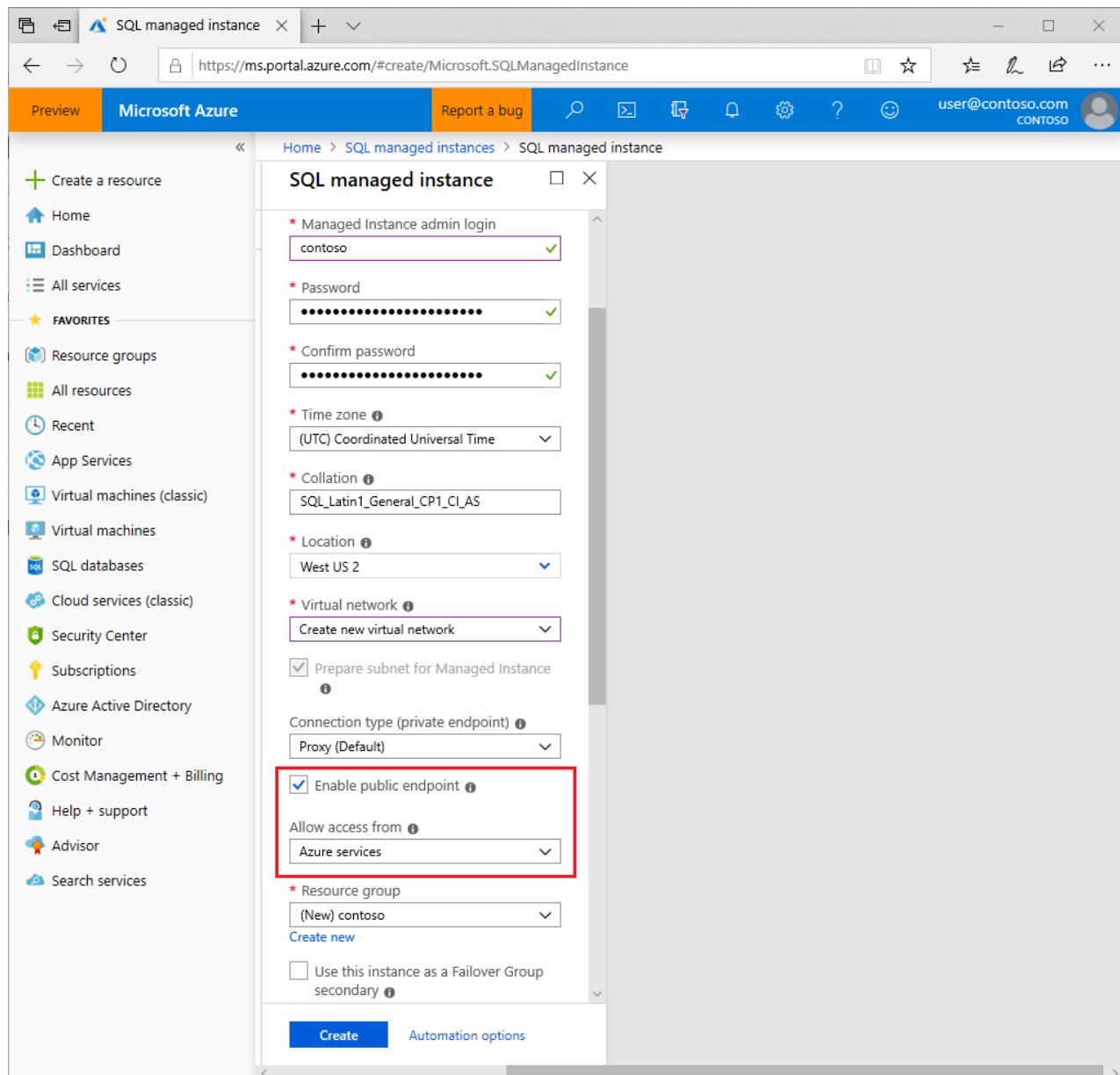
Configure a connection from an Azure Cognitive Search indexer to SQL Managed Instance

10/4/2020 • 2 minutes to read • [Edit Online](#)

As noted in [Connecting Azure SQL Database to Azure Cognitive Search using indexers](#), creating indexers against SQL Managed Instances is supported by Azure Cognitive Search through the public endpoint.

Create Azure SQL Managed Instance with public endpoint

Create a SQL Managed Instance with the **Enable public endpoint** option selected.



The screenshot shows the Microsoft Azure portal interface for creating a new SQL Managed Instance. The left sidebar lists various services like Home, Dashboard, and App Services. The main content area is titled 'SQL managed instance' and contains the following fields:

- Managed Instance admin login: contoso
- Password and Confirm password: both masked with dots
- Time zone: (UTC) Coordinated Universal Time
- Collation: SQL_Latin1_General_CI_AS
- Location: West US 2
- Virtual network: Create new virtual network (dropdown menu)
- Checkboxes:
 - Prepare subnet for Managed Instance
 - Enable public endpoint (checkbox checked and highlighted with a red border)
- Connection type (private endpoint): Proxy (Default)
- Allow access from: Azure services (dropdown menu)
- Resource group: (New) contoso
- Checkboxes:
 - Use this instance as a Failover Group secondary

At the bottom are 'Create' and 'Automation options' buttons.

Enable Azure SQL Managed Instance public endpoint

You can also enable public endpoint on an existing SQL Managed Instance under **Security > Virtual network > Public endpoint > Enable**.

The screenshot shows the Microsoft Azure portal interface. The left sidebar contains a navigation menu with various service icons. The main content area is titled "contoso - Virtual network" under "SQL managed instances". A callout box highlights the "Public endpoint (data)" section, which includes an "Enable" button. Below this, there are fields for "Host" (contoso.public.000000000000.database.windows.net) and "Port" (3342). Under "Connection type (private endpoint)", it says "Proxy (Default)". A note at the top states: "Public endpoint provides the ability to connect to Managed Instance from the Internet without using VPN and is for data communication (TDS) only."

Verify NSG rules

Check the Network Security Group has the correct **Inbound security rules** that allow connections from Azure services.

The screenshot shows the Microsoft Azure portal interface. The left sidebar contains a navigation menu with various service icons. The main content area is titled "nsg-contoso - Inbound security rules" under "Network security group". A callout box highlights the row for "public_endpoint_inbound" with priority 1300, which allows TCP port 3342 from AzureCloud to Any destination. Other rows include "allow_management_inbound", "allow_missubnet_inbound", "allow_health_probe_inbound", "allow_tds_inbound", "allow_redirect_inbound", "allow_geodr_inbound", "deny_all_inbound", "AllowVnetInBound", "AllowAzureLoadBalancerInbound", and "DenyAllInBound".

PRIORITY	NAME	PORT	PROTOCOL	SOURCE	DESTINATION	ACTION
100	allow_management_inbound	9000-9003,14... Any	TCP Any	Any	Any	Allow
200	allow_missubnet_inbound	Any	TCP Any	10.0.0/16	Any	Allow
300	allow_health_probe_inbound	Any	TCP Any	AzureLoadBa...	Any	Allow
1000	allow_tds_inbound	1433	TCP VirtualNetwork	Any	Any	Allow
1100	allow_redirect_inbound	11000-11999	TCP VirtualNetwork	Any	Any	Allow
1200	allow_geodr_inbound	5022	TCP VirtualNetwork	Any	Any	Allow
1300	public_endpoint_inbound	3342	TCP AzureCloud	Any	Any	Allow
4096	deny_all_inbound	Any	TCP Any	Any	Any	Deny
65000	AllowVnetInBound	Any	TCP VirtualNetwork	VirtualNetwork	Any	Allow
65001	AllowAzureLoadBalancerInbound	Any	TCP AzureLoadBal...	AzureLoadBal...	Any	Allow
65500	DenyAllInBound	Any	TCP Any	Any	Any	Deny

NOTE

Indexers still require that SQL Managed Instance be configured with a public endpoint in order to read data. However, you can choose to restrict the inbound access to that public endpoint by replacing the current rule (`public_endpoint_inbound`) with the following 2 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 of those 2 rules, set "PORT" = `3342`, "PROTOCOL" = `TCP`, "DESTINATION" = `Any`, "ACTION" = `Allow`

Get public endpoint connection string

Make sure you use the connection string for the **public endpoint** (port 3342, not port 1433).

The screenshot shows the Microsoft Azure portal interface. The left sidebar contains navigation links like 'Create a resource', 'Home', 'Dashboard', 'All services', 'FAVORITES', 'Resource groups', 'All resources', 'Recent', 'App Services', 'Virtual machines (classic)', 'Virtual machines', 'SQL databases', 'Cloud services (classic)', 'Security Center', 'Subscriptions', 'Azure Active Directory', and 'Monitor'. The main content area is titled 'contoso - Connection strings' and shows a list of connection strings under the 'ADO.NET' tab. One connection string is highlighted with a red box.

Connection String Type	ConnectionString Value
ADO.NET (SQL authentication) - private endpoint	Server=tcp:contoso.0000000000.database.windows.net,1433;Persist Security Info=False;User ID={your_username};Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
ADO.NET (SQL authentication) - public endpoint	Server=tcp:contoso.public.0000000000.database.windows.net,3342;Persist Security Info=False;User ID={your_username};Password={your_password};MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;

Next steps

With configuration out of the way, you can now specify a SQL Managed Instance as the data source for an Azure Cognitive Search indexer using either the portal or REST API. See [Connecting Azure SQL Database to Azure Cognitive Search using indexers](#) for more information.

Configure a connection from an Azure Cognitive Search indexer to SQL Server on an Azure VM

10/4/2020 • 5 minutes to read • [Edit Online](#)

As noted in [Connecting Azure SQL Database to Azure Cognitive Search using indexers](#), creating indexers against **SQL Server on Azure VMs** (or **SQL Azure VMs** for short) is supported by Azure Cognitive Search, but there are a few security-related prerequisites to take care of first.

Connections from Azure Cognitive Search to SQL Server on a VM is a public internet connection. All of the security measures you would normally follow for these connections apply here as well:

- Obtain a certificate from a [Certificate Authority provider](#) for the fully qualified domain name of the SQL Server instance on the Azure VM.
- Install the certificate on the VM, and then enable and configure encrypted connections on the VM using the instructions in this article.

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](#).
 - For VMs created using the newer **Resource Manager** template, the FQDN is formatted as
`<your-VM-name>.<region>.cloudapp.azure.com`
 - For older VMs created as a **Classic** VM, the FQDN is formatted as
`<your-cloud-service-name.cloudapp.net>`.
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 is joined). When names don't match, use regedit to specify the certificate.

- 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.

- Set the value of the **Certificate** key to the **thumbprint** 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 will 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 will not start. You can use the **Certificates** snap-in or **CertUtil**s for this task.

4. Restart the SQL Server service.

Configure SQL Server connectivity in the VM

After you set up the encrypted connection required by Azure Cognitive Search, there are additional configuration steps intrinsic to SQL Server on Azure VMs. If you haven't done so already, the next step is to finish configuration using either one of these articles:

- For a **Resource Manager** VM, see [Connect to a SQL Server Virtual Machine on Azure using Resource Manager](#).
- For a **Classic** VM, see [Connect to a SQL Server Virtual Machine on Azure Classic](#).

In particular, review the section in each article for "connecting over the internet".

Configure the Network Security Group (NSG)

It is not unusual to configure the NSG 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.

NOTE

For background, see [What is a Network Security Group?](#)

- For a **Resource Manager** VM, see [How to create NSGs for ARM deployments](#).
- For a **Classic** VM, see [How to create NSGs for Classic deployments](#).

IP addressing can pose a few challenges that are easily overcome if you are 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.

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.

Managing IP address fluctuations

If your search service has only one search unit (that is, one replica and one partition), the IP address will change during routine service restarts, invalidating an existing ACL with your search service's IP address.

One way to avoid the subsequent connectivity error is to use more than one replica and one partition in Azure Cognitive Search. Doing so increases the cost, but it also solves the IP address problem. In Azure Cognitive Search, IP addresses don't change when you have more than one search unit.

A second approach is to allow the connection to fail, and then reconfigure the ACLs in the NSG. On average, you can expect IP addresses to change every few weeks. For customers who do controlled indexing on an infrequent basis, this approach might be viable.

A third viable (but not particularly secure) approach is to specify the IP address range of the Azure region where your search service is provisioned. The list of IP ranges from which public IP addresses are allocated to Azure resources is published at [Azure Datacenter IP ranges](#).

Include the Azure Cognitive Search portal IP addresses

If you are using the Azure portal to create an indexer, Azure Cognitive Search portal logic also needs access to your SQL Azure VM during creation time. Azure Cognitive Search portal IP addresses can be found by pinging `stamp2.search.ext.azure.com`.

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. See [Connecting Azure SQL Database to Azure Cognitive Search using indexers](#) for more information.

Attach a Cognitive Services resource to a skillset in Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

When configuring an 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 Cognitive Services resource.

In this article, you'll learn how to attach a resource by assigning a key to a skillset that defines an enrichment pipeline.

Resources used during enrichment

Azure Cognitive Search has a dependency on Cognitive Services, including [Computer Vision](#) for image analysis and optical character recognition (OCR), [Text Analytics](#) for natural language processing, and other enrichments like [Text Translation](#). In the context of enrichment in Azure Cognitive Search, these AI algorithms are wrapped inside a *skill*, placed in a *skillset*, and referenced by an *indexer* during indexing.

How billing works

- Azure Cognitive Search uses the Cognitive Services resource key you provide on a skillset to bill for image and text enrichment. Execution of billable skills is at the [Cognitive Services pay-as-you go price](#).
- Image extraction is an Azure Cognitive Search operation that occurs when documents are cracked prior to enrichment. Image extraction is billable. For image extraction pricing, see the [Azure Cognitive Search pricing page](#).
- Text extraction also occurs during the document cracking phrase. It is not billable.
- Skills that do not call Cognitive Services, including Conditional, Shaper, Text Merge, and Text Split skills, are not billable.

Same-region requirement

We require that Azure Cognitive Search and Azure Cognitive Services exist within the same region. Otherwise, you will get this message at run time:

"Provided key is not a valid CognitiveServices type key for the region of your search service."

There is no way to move a service across regions. If you get this error, you should create a new Cognitive Services resource in the same region as Azure Cognitive Search.

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 non-regional services, see the [Cognitive Services product by region](#) page.

Use Free resources

You can use a limited, free processing option to complete the AI enrichment tutorial and quickstart exercises.

Free (Limited enrichments) resources are restricted to 20 documents per day, per indexer. You can delete and recreate the indexer to reset the counter.

1. Open the Import data wizard:



2. Choose a data source and continue to **Add AI enrichment (Optional)**. For a step-by-step walkthrough of this wizard, see [Create an index in the Azure portal](#).
3. Expand **Attach Cognitive Services** and then select **Free (Limited enrichments)**:

A screenshot of the 'Import data' wizard. At the top, there are tabs: 'Connect to your data' (selected), 'Add cognitive search (Optional)', 'Customize target index', and 'Create an indexer'. Below the tabs, there's an information box with an 'i' icon: 'Enrich and extract structure from your documents through cognitive skills using the same AI algorithms that power Cognitive Services. Select the document cracking options and the cognitive skills you want to apply to your documents.' A link 'Learn more' is provided. Underneath, there's a section titled 'Attach Cognitive Services' with a dropdown arrow. A note says: 'To power your cognitive skills, select an existing Cognitive Services resource or create a new one. The Cognitive Services resource should be in the same region as your Search service. The execution of cognitive skills will be billed to the selected resource. Otherwise, the number of enrichments executions will be limited.' Another link 'Learn more' is shown. Below this, there's a table with columns 'COGNITIVE SERVICES RESOURCE NAME' and 'REGION'. A row shows 'Free (Limited enrichments)' under 'COGNITIVE SERVICES RESOURCE NAME' and a blue bar under 'REGION'. A 'Refresh' button is at the top right of the table area. At the bottom left, there's a link 'Create new Cognitive Services resource'. At the very bottom, there's a collapsed section 'Add Enrichments' with a minus sign.

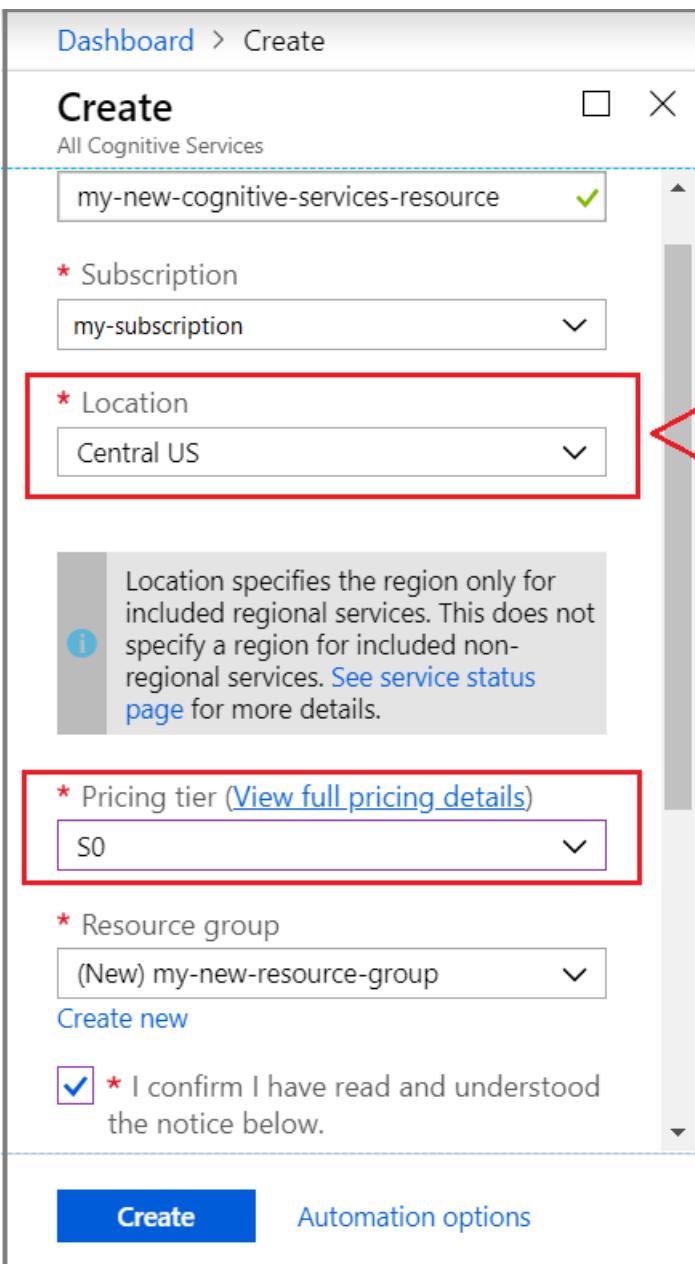
4. You can now continue on to the next steps, including **Add cognitive skills**.

Use billable resources

For workloads that create more than 20 enrichments per day, make sure to attach a billable Cognitive Services resource. We recommend that you always attach a billable Cognitive Services resource, even if you never intend to call Cognitive Services APIs. Attaching a resource overrides the daily limit.

You're charged only for skills that call the Cognitive Services APIs. You're not billed for [custom skills](#), or skills like [text merger](#), [text splitter](#), and [shaper](#), which aren't API-based.

1. Open the Import data wizard, choose a data source, and continue to **Add AI enrichment (Optional)**.
2. Expand **Attach Cognitive Services** and then select **Create new Cognitive Services resource**. A new tab opens so that you can create the resource:



3. In the **Location** list, select the region where your Azure Cognitive Search service is located. Make sure to use this region for performance reasons. Using this region also voids outbound bandwidth charges across regions.
4. In the **Pricing tier** list, select **S0** to get the all-in-one collection of Cognitive Services features, including the Vision and Language features that back the built-in skills provided by Azure Cognitive Search.

For the S0 tier, you can find rates for specific workloads on the [Cognitive Services pricing page](#).

 - In the **Select Offer** list, make sure **Cognitive Services** is selected.
 - Under **Language** features, the rates for **Text Analytics Standard** apply to AI indexing.
 - Under **Vision** features, the rates for **Computer Vision S1** apply.
5. Select **Create** to provision the new Cognitive Services resource.
6. Return to the previous tab, which contains the Import data wizard. Select **Refresh** to show the Cognitive Services resource, and then select the resource:

^ Attach Cognitive Services

To power your cognitive skills, select an existing Cognitive Services resource or create a new one. The Cognitive Services resource should be in the same region as your Search service. The execution of cognitive skills will be billed to the selected resource. Otherwise, the number of enrichments executions will be limited. [Learn more](#)

[Refresh](#)

COGNITIVE SERVICES RESOURCE NAME

REGION

Free (Limited enrichments)

my-cog-services-resource

westus2

[Create new Cognitive Services resource](#)

7. Expand the **Add cognitive skills** section to select the specific cognitive skills that you want to run on your data. Complete the rest of the wizard.

Attach an existing skillset to a Cognitive Services resource

If you have an existing skillset, you can attach it to a new or different Cognitive Services resource.

1. On the **Service overview** page, select **Skillsets**:

The screenshot shows the 'Skillsets' tab selected in the top navigation bar. Below the tabs, there is a table with two columns: 'NAME' and 'NUMBER OF SKILLS'. A single row is visible, containing 'myskillset' in the NAME column and '1' in the NUMBER OF SKILLS column. The entire row is highlighted with a red box.

NAME	NUMBER OF SKILLS
myskillset	1

2. Select the name of the skillset, and then select an existing resource or create a new one. Select **OK** to confirm your changes.

The screenshot shows the configuration dialog for the 'myskillset' skillset. At the top, there is a title bar with the skillset name and standard window controls. Below the title bar, there is a delete button labeled 'Delete skillset'. The main content area contains instructions for selecting a Cognitive Services resource and a note about billing. At the bottom, there is a 'Refresh' button and a table with 'COGNITIVE SERVICES RESOURCE NAME' and 'REGION' columns, showing 'my-cog-services-resource' and 'westus2' respectively. A blue dashed box highlights the 'my-cog-services-resource' entry.

COGNITIVE SERVICES RESOURCE NAME	REGION
my-cog-services-resource	westus2

Remember that the **Free (Limited enrichments)** option limits you to 20 documents daily, and that you can use **Create new Cognitive Services resource** to provision a new billable resource. If you create a new resource, select **Refresh** to refresh the list of Cognitive Services resources, and then select the

resource.

Attach Cognitive Services programmatically

When you're defining the skillset programmatically, add a `cognitiveServices` section to the skillset. In that section, include the key of the Cognitive Services resource that you want to associate with the skillset.

Remember that the resource must be in the same region as your Azure Cognitive Search resource. Also include `@odata.type`, and set it to `#Microsoft.Azure.Search.CognitiveServicesByKey`.

The following example shows this pattern. Notice the `cognitiveServices` section at the end of the definition.

```
PUT https://[servicename].search.windows.net/skillsets/[skillset name]?api-version=2020-06-30
api-key: [admin key]
Content-Type: application/json
```

```
{
  "name": "skillset name",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
      "categories": [ "Organization" ],
      "defaultLanguageCode": "en",
      "inputs": [
        {
          "name": "text", "source": "/document/content"
        }
      ],
      "outputs": [
        {
          "name": "organizations", "targetName": "organizations"
        }
      ]
    },
    "cognitiveServices": {
      "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",
      "description": "mycogsvcs",
      "key": "<your key goes here>"
    }
  }
}
```

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 equals 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](#)

How to create a skillset in an AI enrichment pipeline in Azure Cognitive Search

10/4/2020 • 8 minutes to read • [Edit Online](#)



A skillset defines the operations that extract and enrich data to make it searchable. A skillset executes after text and image content are extracted from source documents, and after any fields from the source document are (optionally) mapped to destination fields in an index or knowledge store.

A skillset contains one or more *cognitive skills* that represent a specific enrichment operation, like translating text, extracting key phrases, or performing optical character recognition from an image file. To create a skillset, you can use [built-in skills](#) from Microsoft, or custom skills that contain models or processing logic that you provide (see [Example: Creating a custom skill in an AI enrichment pipeline](#) for more information).

In this article, you learn how to create an enrichment pipeline for the skills you want to use. A skillset is attached to an Azure Cognitive Search [indexer](#). One part of pipeline design, covered in this article, is constructing the skillset itself.

NOTE

Another part of pipeline design is specifying an indexer, covered in the [next step](#). An indexer definition includes a reference to the skillset, plus field mappings used for connecting inputs to outputs in the target index.

Key points to remember:

- You can only have one skillset per indexer.
- A skillset must have at least one skill.
- You can create multiple skills of the same type (for example, variants of an image analysis skill).

Begin with the end in mind

A recommended initial step is deciding which data to extract from your raw data and how you want to use that data in a search solution. Creating an illustration of the entire enrichment pipeline can help you identify the necessary steps.

Suppose you are interested in processing a set of financial analyst comments. For each file, you want to extract company names and the general sentiment of the comments. You might also want to write a custom enricher that uses the Bing Entity Search service to find additional information about the company, such as what kind of business the company is engaged in. Essentially, you want to extract information like the following, indexed for each document:

RECORD-TEXT

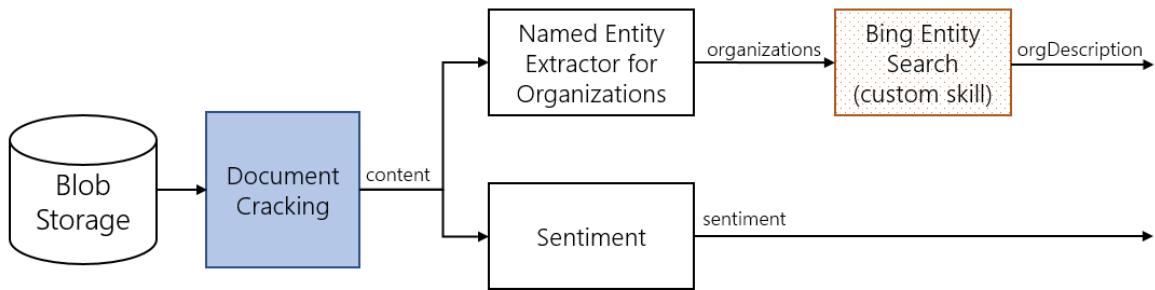
COMPANIES

SENTIMENT

COMPANY DESCRIPTIONS

RECORD-TEXT	COMPANIES	SENTIMENT	COMPANY DESCRIPTIONS
sample-record	["Microsoft", "LinkedIn"]	0.99	["Microsoft Corporation is an American multinational technology company ...", "LinkedIn is a business- and employment-oriented social networking..."]

The following diagram illustrates a hypothetical enrichment pipeline:



Once you have fair idea of what you want in the pipeline, you can express the skillset that provides these steps. Functionally, the skillset is expressed when you upload your indexer definition to Azure Cognitive Search. To learn more about how to upload your indexer, see the [indexer-documentation](#).

In the diagram, the *document cracking* step happens automatically. Essentially, Azure Cognitive Search knows how to open well-known files and creates a *content* field containing the text extracted from each document. The white boxes are built-in enrichers, and the dotted "Bing Entity Search" box represents a custom enricher that you are creating. As illustrated, the skillset contains three skills.

Skillset definition in REST

A skillset is defined as an array of skills. Each skill defines the source of its inputs and the name of the outputs produced. Using the [Create Skillset REST API](#), you can define a skillset that corresponds to the previous diagram:

```

PUT https://[servicename].search.windows.net/skillsets/[skillset name]?api-version=2020-06-30
api-key: [admin key]
Content-Type: application/json
  
```

```
{
  "description": 
    "Extract sentiment from financial records, extract company names, and then find additional information about each company mentioned.",
  "skills": 
  [
    {
      "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
      "context": "/document",
      "categories": [ "Organization" ],
      "defaultLanguageCode": "en",
      "inputs": [
        {
          "name": "text",
          "source": "/document/content"
        }
      ],
      "outputs": [
        {
          "name": "organizations",
          "targetName": "organizations"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
      "inputs": [
        {
          "name": "text",
          "source": "/document/content"
        }
      ],
      "outputs": [
        {
          "name": "score",
          "targetName": "mySentiment"
        }
      ]
    },
    {
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
      "description": "Calls an Azure function, which in turn calls Bing Entity Search",
      "uri": "https://indexer-e2e-webskill.azurewebsites.net/api/InvokeTextAnalyticsV3?code=foo",
      "httpHeaders": {
        "Ocp-Apim-Subscription-Key": "foobar"
      },
      "context": "/document/organizations/*",
      "inputs": [
        {
          "name": "query",
          "source": "/document/organizations/*"
        }
      ],
      "outputs": [
        {
          "name": "description",
          "targetName": "companyDescription"
        }
      ]
    }
  ]
}
```

Create a skillset

While creating a skillset, you can provide a description to make the skillset self-documenting. A description is optional, but useful for keeping track of what a skillset does. Because skillset is a JSON document, which does not allow comments, you must use a `description` element for this.

```
{  
  "description":  
    "This is our first skill set, it extracts sentiment from financial records, extract company names,  
    and then finds additional information about each company mentioned.",  
  ...  
}
```

The next piece in the skillset is an array of skills. You can think of each skill as a primitive of enrichment. Each skill performs a small task in this enrichment pipeline. Each one takes an input (or a set of inputs), and returns some outputs. The next few sections focus on how to specify built-in and custom skills, chaining skills together through input and output references. Inputs can come from source data or from another skill. Outputs can be mapped to a field in a search index or used as an input to a downstream skill.

Add built-in skills

Let's look at the first skill, which is the built-in [entity recognition skill](#):

```
{  
  "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",  
  "context": "/document",  
  "categories": [ "Organization" ],  
  "defaultLanguageCode": "en",  
  "inputs": [  
    {  
      "name": "text",  
      "source": "/document/content"  
    }  
  ],  
  "outputs": [  
    {  
      "name": "organizations",  
      "targetName": "organizations"  
    }  
  ]  
}
```

- Every built-in skill has `odata.type`, `input`, and `output` properties. Skill-specific properties provide additional information applicable to that skill. For entity recognition, `categories` is one entity among a fixed set of entity types that the pretrained model can recognize.
- Each skill should have a `"context"`. The context represents the level at which operations take place. In the skill above, the context is the whole document, meaning that the entity recognition skill is called once per document. Outputs are also produced at that level. More specifically, `"organizations"` are generated as a member of `"/document"`. In downstream skills, you can refer to this newly created information as `"/document/organizations"`. If the `"context"` field is not explicitly set, the default context is the document.
- The skill has one input called "text", with a source input set to `"/document/content"`. The skill (entity recognition) operates on the `content` field of each document, which is a standard field created by the Azure blob indexer.
- The skill has one output called `"organizations"`. Outputs exist only during processing. To chain this output to a downstream skill's input, reference the output as `"/document/organizations"`.

- For a particular document, the value of `"/document/organizations"` is an array of organizations extracted from the text. For example:

```
["Microsoft", "LinkedIn"]
```

Some situations call for referencing each element of an array separately. For example, suppose you want to pass each element of `"/document/organizations"` separately to another skill (such as the custom Bing entity search enricher). You can refer to each element of the array by adding an asterisk to the path:

```
"/document/organizations/*"
```

The second skill for sentiment extraction follows the same pattern as the first enricher. It takes `"/document/content"` as input, and returns a sentiment score for each content instance. Since you did not set the `"context"` field explicitly, the output (mySentiment) is now a child of `"/document"`.

```
{
  "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "score",
      "targetName": "mySentiment"
    }
  ]
},
```

Add a custom skill

Recall the structure of the custom Bing entity search enricher:

```
{
  "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
  "description": "This skill calls an Azure function, which in turn calls Bing Entity Search",
  "uri": "https://indexer-e2e-webskill.azurewebsites.net/api/InvokeTextAnalyticsV3?code=foo",
  "httpHeaders": {
    "Ocp-Apim-Subscription-Key": "foobar"
  },
  "context": "/document/organizations/*",
  "inputs": [
    {
      "name": "query",
      "source": "/document/organizations/*"
    }
  ],
  "outputs": [
    {
      "name": "description",
      "targetName": "companyDescription"
    }
  ]
}
```

This definition is a [custom skill](#) that calls a web API as part of the enrichment process. For each organization identified by entity recognition, this skill calls a web API to find the description of that organization. The orchestration of when to call the web API and how to flow the information received is handled internally by

the enrichment engine. However, the initialization necessary for calling this custom API must be provided in the JSON (such as uri, httpHeaders, and the inputs expected). For guidance in creating a custom web API for the enrichment pipeline, see [How to define a custom interface](#).

Notice that the "context" field is set to `"/document/organizations/*"` with an asterisk, meaning the enrichment step is called *for each* organization under `"/document/organizations"`.

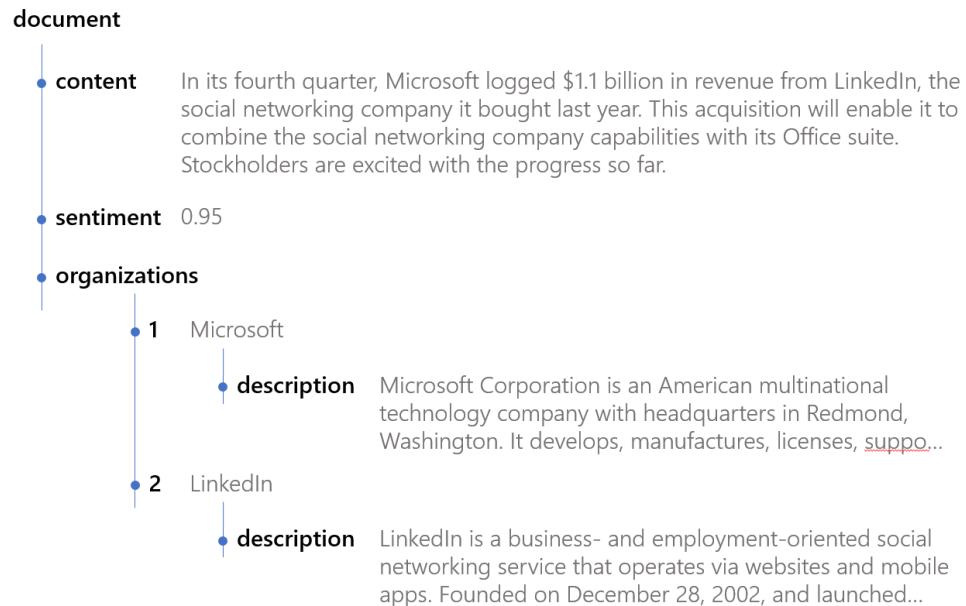
Output, in this case a company description, is generated for each organization identified. When referring to the description in a downstream step (for example, in key phrase extraction), you would use the path `"/document/organizations/*/description"` to do so.

Add structure

The skillset generates structured information out of unstructured data. Consider the following example:

"In its fourth quarter, Microsoft logged \$1.1 billion in revenue from LinkedIn, the social networking company it bought last year. The acquisition enables Microsoft to combine LinkedIn capabilities with its CRM and Office capabilities. Stockholders are excited with the progress so far."

A likely outcome would be a generated structure similar to the following illustration:



Until now, this structure has been internal-only, memory-only, and used only in Azure Cognitive Search indexes. The addition of a knowledge store gives you a way to save shaped enrichments for use outside of search.

Add a knowledge store

[Knowledge store](#) is a feature in Azure Cognitive Search for saving your enriched document. A knowledge store that you create, backed by an Azure storage account, is the repository where your enriched data lands.

A knowledge store definition is added to a skillset. For a walkthrough of the entire process, see [Create a knowledge store in REST](#).

```
"knowledgeStore": {  
    "storageConnectionString": "<an Azure storage connection string>",  
    "projections" : [  
        {  
            "tables": [ ]  
        },  
        {  
            "objects": [  
                {  
                    "storageContainer": "containername",  
                    "source": "/document/EnrichedShape/",  
                    "key": "/document/Id"  
                }  
            ]  
        }  
    ]  
}
```

You can choose to save the enriched documents as tables with hierarchical relationships preserved or as JSON documents in blob storage. Output from any of the skills in the skillset can be sourced as the input for the projection. If you are looking to project the data into a specific shape, the updated [shaper skill](#) can now model complex types for you to use.

Next steps

Now that you are familiar with the enrichment pipeline and skillsets, continue with [How to reference annotations in a skillset](#) or [How to map outputs to fields in an index](#).

How to reference annotations in an Azure Cognitive Search skillset

10/4/2020 • 4 minutes to read • [Edit Online](#)

In this article, you learn how to reference annotations in skill definitions, using examples to illustrate various scenarios. As the content of a document flows through a set of skills, it gets enriched with annotations. Annotations can be used as inputs for further downstream enrichment, or mapped to an output field in an index.

Examples in 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*.

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 created and used by the pipeline to hold 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.
Enrichment Context	The context in which the enrichment takes place, in terms of which element is enriched. By default, the enrichment context is at the <code>"/document"</code> level, scoped to individual documents. When a skill runs, the outputs of that skill become properties of the defined context .

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.

```
{
  "@odata.type": "#Microsoft.Skills.Text.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 is called for each member within the document until all members are processed.

```
{
  "@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 is 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.

```
{  
    "@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"  
        }  
    ]  
}
```

See also

- [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](#)

How to map AI-enriched fields to a searchable index

10/4/2020 • 3 minutes to read • [Edit Online](#)



In this article, you learn how to map enriched input fields to output fields in a searchable index. Once you have [defined a skillset](#), you must map the output fields of any skill that directly contributes values to a given field in your search index.

Output Field Mappings are required for moving content from enriched documents into the index. The enriched document is really a tree of information, and even though there is support for complex types in the index, sometimes you may want to transform the information from the enriched tree into a more simple type (for instance, an array of strings). Output field mappings allow you to perform data shape transformations by flattening information. Output field mappings always occur after skillset execution, although it is possible for this stage to run even if no skillset is defined.

Examples of output field mappings:

- As part of your skillset, you extracted the names of organizations mentioned in each of the pages of your document. Now you want to map each of those organization names into a field in your index of type Edm.Collection(Edm.String).
- As part of your skillset, you produced a new node called "document/translated_text". You would like to map the information on this node to a specific field in your index.
- You don't have a skillset but are indexing a complex type from a Cosmos DB database. You would like to get to a node on that complex type and map it into a field in your index.

NOTE

We recently enabled the functionality of mapping functions on output field mappings. For more details on mapping functions, see [Field mapping functions](#)

Use `outputFieldMappings`

To map fields, add `outputFieldMappings` to your indexer definition as shown below:

```
PUT https://[servicename].search.windows.net/indexers/[indexer name]?api-version=2020-06-30
api-key: [admin key]
Content-Type: application/json
```

The body of the request is structured as follows:

```
{
  "name": "myIndexer",
  "dataSourceName": "myDataSource",
  "targetIndexName": "myIndex",
  "skillsetName": "myFirstSkillSet",
  "fieldMappings": [
    {
      "sourceFieldName": "metadata_storage_path",
      "targetFieldName": "id",
      "mappingFunction": {
        "name": "base64Encode"
      }
    }
  ],
  "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).

Flattening Information from Complex Types

The path in a sourceFieldName can represent one element or multiple elements. In the example above,

`/document/content/sentiment` represents a single numeric value, while

`/document/content/organizations/*/description` represents several organization descriptions.

In cases where there are several elements, they are "flattened" into an array that contains each of the elements.

More concretely, for the `/document/content/organizations/*/description` example, the data in the *descriptions* field would look like a flat array of descriptions before it gets indexed:

```
["Microsoft is a company in Seattle","LinkedIn's office is in San Francisco"]
```

This is an important principle, so we will provide another example. Imagine that you have an array of complex types as part of the enrichment tree. Let's say there is a member called customEntities that has an array of complex types like the one described below.

```

"document/customEntities":
[
  {
    "name": "heart failure",
    "matches": [
      {
        "text": "heart failure",
        "offset": 10,
        "length": 12,
        "matchDistance": 0.0
      }
    ]
  },
  {
    "name": "morquio",
    "matches": [
      {
        "text": "morquio",
        "offset": 25,
        "length": 7,
        "matchDistance": 0.0
      }
    ]
  }
  //...
]

```

Let's assume that your index has a field called 'diseases' of type Collection(Edm.String), where you would like to store each of the names of the entities.

This can be done easily by using the "*" symbol, as follows:

```

"outputFieldMappings": [
  {
    "sourceFieldName": "/document/customEntities/*/name",
    "targetFieldName": "diseases"
  }
]

```

This operation will simply "flatten" each of the names of the customEntities elements into a single array of strings like this:

```

"diseases" : ["heart failure","morquio"]

```

Next steps

Once you have mapped your enriched fields to searchable fields, you can set the field attributes for each of the searchable fields [as part of the index definition](#).

For more information about field mapping, see [Field mappings in Azure Cognitive Search indexers](#).

How to process and extract information from images in AI enrichment scenarios

10/4/2020 • 7 minutes to read • [Edit Online](#)

Azure Cognitive Search has several capabilities for working with images and image files. During document cracking, you can use the *imageAction* parameter to extract text from photos or pictures containing alphanumeric text, such as the word "STOP" in a stop sign. Other scenarios include generating 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 image processing in more detail and provides guidance for working with images in an AI enrichment pipeline.

Get normalized images

As part of document cracking, there are a new set of indexer configuration parameters for handling image files or images embedded in files. These parameters are used to normalize images for further downstream processing. Normalizing images makes them more uniform. Large images are resized to a maximum height and width to make them consumable. For images providing 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 expect normalized images. Enabling image normalization on an indexer requires that a skillset be attached to that indexer.

CONFIGURATION PARAMETER	DESCRIPTION
imageAction	<p>Set to "none" if no action should be taken when embedded images or image files are encountered.</p> <p>Set to "generateNormalizedImages" to generate an array of normalized images as part of document cracking.</p> <p>Set to "generateNormalizedImagePerPage" to generate an array of normalized images where, for PDFs in your data source, each page is rendered to one output image. The functionality is the same as "generateNormalizedImages" for non-PDF file types.</p> <p>For any option that is not "none", the images will be exposed in the <i>normalized_images</i> field.</p> <p>The default is "none." This configuration is only pertinent to blob data sources, when "dataToExtract" is set to "contentAndMetadata."</p> <p>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.</p>
normalizedImageMaxWidth	<p>The maximum width (in pixels) for normalized images generated. The default is 2000. The maximum value allowed is 10000.</p>
normalizedImageMaxHeight	<p>The maximum height (in pixels) for normalized images generated. The default is 2000. The maximum value allowed is 10000.</p>

NOTE

If you set the `imageAction` property to anything other than "none", you'll not be able to set the `parsingMode` property to anything other than "default". You may only set one of these two properties to a non-default value in your indexer configuration.

Set the `parsingMode` parameter to `json` (to index each blob as a single document) or `jsonArray` (if your blobs contain JSON arrays and you need each element of an array to be treated as a separate document).

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.

You specify the `imageAction` in your [indexer definition](#) as follows:

```
{  
    //...rest of your indexer definition goes here ...  
    "parameters":  
    {  
        "configuration":  
        {  
            "dataToExtract": "contentAndMetadata",  
            "imageAction": "generateNormalizedImages"  
        }  
    }  
}
```

When the `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.
contentOffset	The character offset within the content field where the image was extracted from. This field is only applicable for files with embedded images.

IMAGE MEMBER	DESCRIPTION
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`:

```
[
  {
    "data": "BASE64 ENCODED STRING OF A JPEG IMAGE",
    "width": 500,
    "height": 300,
    "originalWidth": 5000,
    "originalHeight": 3000,
    "rotationFromOriginal": 90,
    "contentOffset": 500,
    "pageNumber": 2
  }
]
```

Image-related skills

There are two built-in cognitive skills that take images as an input: [OCR](#) and [Image Analysis](#).

Currently, these skills only work with images generated from the document cracking step. As such, the only supported input is `"/document/normalized_images"`.

Image Analysis skill

The [Image Analysis skill](#) extracts a rich set of visual features based on the image content. For instance, you can generate a caption from an image, generate tags, or identify celebrities and landmarks.

OCR skill

The [OCR skill](#) extracts text from image files such as JPGs, PNGs, and bitmaps. It can extract text as well as layout information. The layout information provides bounding boxes for each of the strings identified.

Embedded image scenario

A common scenario involves creating a single string containing all file contents, both text and image-origin text, by performing the following steps:

1. [Extract normalized_images](#)
2. Run the OCR skill using `"/document/normalized_images"` as input
3. Merge the text representation of those images with the raw text extracted from the file. You can use the [Text Merge](#) skill to consolidate both text chunks into a single large string.

The following example skillset creates a `merged_text` field containing the textual content of your document. It also includes the OCRed text from each of the embedded images.

Request body syntax

```
{
  "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"
          }
        ]
      }
    ]
  }
}
```

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.

Visualize bounding boxes of extracted text

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:

```
/// <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;
}
```

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 configure caching for incremental enrichment in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

IMPORTANT

Incremental enrichment is currently in public preview. This preview version is provided without a service level agreement, and it's not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). REST API preview versions provide this feature. There is no portal or .NET SDK support at this time.

This article shows you how to add caching to an enrichment pipeline so that you can incrementally modify steps without having to rebuild every time. By default, a skillset is stateless, and changing any part of its composition requires a full rerun of the indexer. With incremental enrichment, the indexer can determine which parts of the document tree need to 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-numeric-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](#).

Enable caching on an existing indexer

If you have an existing indexer that already has a skillset, follow the steps in this section to add caching. As a one-time operation, you will have to reset and rerun the indexer in full before incremental processing can take effect.

TIP

As proof-of-concept, you can run through this [portal quickstart](#) to create necessary objects, and then use Postman or the portal to make your updates. You might want to attach a billable Cognitive Services resource. Running the indexer multiple times will exhaust the free daily allocation before you can complete all of the steps.

Step 1: Get the indexer definition

Start with a valid, existing indexer that has these components: data source, skillset, index. Your indexer should be runnable.

Using an API client, construct a [GET Indexer request](#) to get the current configuration of the indexer. When you use the preview API version to the GET the indexer, a `cache` property set to null is added to the definitions.

```
GET https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [YOUR-ADMIN-KEY]
```

Copy the indexer definition from the response.

Step 2: Modify the cache property in the indexer definition

By default the `cache` property is null. Use an API client to set the cache configuration (the portal does not support this particular update).

Modify the cache object to include the following required and optional properties:

- The `storageConnectionString` is required, and it must be set to an Azure storage connection string.
- The `enableReprocessing` boolean property is optional (`true` by default), and it indicates that incremental enrichment is enabled. When needed, you can set it to `false` to suspend incremental processing while other resource-intensive operations, such as indexing new documents, are underway and then flip it back to `true` later.

```
{  
    "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

A reset of the 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 and the [Reset Indexer REST API](#) for this task.

```
POST https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]/reset?api-version=2020-06-30-Preview  
Content-Type: application/json  
api-key: [YOUR-ADMIN-KEY]
```

Step 4: Save the updated definition

Update the indexer with a PUT request, the body of the request should contain the updated indexer definition that has the cache property. If you get a 400, check the indexer definition to make sure all requirements are met (data source, skillset, index).

```
PUT https://[YOUR-SEARCH-SERVICE].search.windows.net/indexers/[YOUR-INDEXER-NAME]?api-version=2020-06-30-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.

```
"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](#):

```
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]
```

After the indexer runs, you can find the cache in Azure Blob storage. The container name is in the following format: `ms-az-search-indexercache-<YOUR-CACHE-ID>`

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.

Step 6: Modify a skillset and confirm incremental enrichment

To modify a skillset, you can use the portal or the API. For example, if you are using text translation, a simple inline change from `en` to `es` or another language is sufficient for proof-of-concept testing of incremental enrichment.

Run the indexer again. Only those parts of an enriched document tree are updated. If you used the [portal quickstart](#) as proof-of-concept, modifying the text translation skill to 'es', you'll notice that only 8 documents are updated instead of the original 14. Image files unaffected by the translation process are reused from cache.

Enable caching on new indexers

To set up incremental enrichment for a new indexer, all you have to do is include the `cache` property in the indexer definition payload when calling [Create Indexer \(2020-06-30-Preview\)](#).

```
{
    "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": []
}
```

Checking for cached output

The cache is created, used, and managed by the indexer, and its contents are not represented in a format that is human readable. The best way to determine whether the cache is used is by running the indexer and compare before-and-after metrics for execution time and document counts.

For example, assume a skillset that starts with image analysis and Optical Character Recognition (OCR) of scanned documents, followed by downstream analysis of the resulting text. If you modify a downstream text skill, the indexer can retrieve all of the previously processed image and OCR content from cache, updating and processing just text-related changes indicated by your edits. You can expect to see fewer documents in the document count (for example 8/8 as opposed to 14/14 in the original run), shorter execution times, and fewer charges on your bill.

Working with the cache

Once the cache is operational, indexers check the cache whenever [Run Indexer](#) is called, to see which parts of the existing output can be used.

The following table summarizes how various APIs relate to the cache:

API	CACHE IMPACT
Create Indexer (2020-06-30-Preview)	Creates and runs an indexer on first use, including creating a cache if your indexer definition specifies it.
Run Indexer	Executes an enrichment pipeline on demand. This API reads from the cache if it exists, or creates a cache if you added caching to an updated indexer definition. When you run an indexer that has caching enabled, the indexer omits steps if cached output can be used. You can use the generally available or preview API version of this API.
Reset Indexer	Clears the indexer of any incremental indexing information. The next indexer run (either on-demand or schedule) is full reprocessing from scratch, including re-running all skills and rebuilding the cache. It is functionally equivalent to deleting the indexer and recreating it. You can use the generally available or preview API version of this API.
Reset Skills	Specifies which skills to rerun on the next indexer run, even if you haven't modified any skills. The cache is updated accordingly. Outputs, such as a knowledge store or search index, are refreshed using reusable data from the cache plus new content per the updated skill.

For more information about controlling what happens to the cache, see [Cache management](#).

Next steps

Incremental enrichment is applicable on indexers that contain skillsets. As a next step, visit the skillset documentation to understand concepts and composition.

Additionally, once you enable the cache, you will want to know about the parameters and APIs that factor into caching, including how to override or force particular behaviors. For more information, see the following links.

- [Skillset concepts and composition](#)
- [How to create a skillset](#)

- Introduction to incremental enrichment and caching

How to add a custom skill to an Azure Cognitive Search enrichment pipeline

10/4/2020 • 4 minutes to read • [Edit Online](#)

An [enrichment pipeline](#) in Azure Cognitive Search can be assembled from [built-in cognitive skills](#) as well as [custom skills](#) that you personally create and add to the pipeline. In this article, learn how to create a custom skill that exposes an interface allowing it to be included in an AI enrichment pipeline.

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 define field-specific custom entities, 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](#).

Whatever custom capability you require, there is a simple and clear interface for connecting a custom skill to the rest of the enrichment pipeline. The only requirement for inclusion in a [skillset](#) is the ability to accept inputs and emit outputs in ways that are consumable within the skillset as a whole. The focus of this article is on the input and output formats that the enrichment pipeline requires.

Web API custom skill interface

Custom WebAPI skill endpoints by default timeout if they don't return a response 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 window. It is possible to configure the timeout to be up to 230 seconds, by setting the timeout parameter:

```
"@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
"description": "This skill has a 230 second timeout",
"uri": "https://[your custom skill uri goes here]",
"timeout": "PT230S",
```

Make sure the URI is secure (HTTPS).

Currently, the only mechanism for interacting with a custom skill is through a Web API interface. The Web API needs must meet the requirements described in this section.

1. Web API Input Format

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 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.

To be more concrete, per the example above, your Web API should expect requests that look like this:

```
{  
    "values": [  
        {  
            "recordId": "a1",  
            "data":  
                {  
                    "contractText":  
                        "This is a contract that was issued 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 reality, your service may get called with hundreds or thousands of records instead of only the three shown here.

2. Web API Output Format

The format of the output is a set of records containing a *recordId*, and a property bag

```
{
  "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"} ]
    }
  ]
}
```

This particular example has only one output, but you could output more than one property.

Errors and Warning

As shown in the previous example, you may return error and warning messages for each record.

Consuming custom skills from 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 may be described as part of the skillset definition. HTTP headers are not a requirement, but they allow you to add additional configuration capabilities to your skill and to set them from the skillset definition.

```
{  
  "skills": [  
    {  
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
      "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"  
        }  
      ]  
    }  
  ]  
}
```

Next steps

This article covered the interface requirements necessary for integrating a custom skill into a skillset. Click the following links to learn more about custom skills and skillset composition.

- [Watch our video about custom skills](#)
- [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](#)

Example: Create a custom skill using Python

10/4/2020 • 6 minutes to read • [Edit Online](#)

In this Azure Cognitive Search skillset example, you will learn how to create a web API custom skill using Python and Visual Studio Code. The example uses an [Azure Function](#) that implements the [custom skill interface](#).

The custom skill is simple by design (it concatenates two strings) so that you can focus on the tools and technologies used for custom skill development in Python. Once you succeed with a simple skill, you can branch out with more complex scenarios.

Prerequisites

- Review the [custom skill interface](#) for an introduction into the input/output interface that a custom skill should implement.
- Set up your environment. We followed [this tutorial end-to-end](#) to set up serverless Azure Function using Visual Studio Code and Python extensions. The tutorial leads you through installation of the following tools and components:
 - [Python 3.75](#)
 - [Visual Studio Code](#)
 - [Python extension for Visual Studio Code](#)
 - [Azure Functions Core Tools](#)
 - [Azure Functions extension for Visual Studio Code](#)

Create an Azure Function

This example uses an Azure Function to demonstrate the concept of hosting a web API, but other approaches are possible. 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

The Azure Functions project template in Visual Studio Code creates a project that can be published to a function app in Azure. A function app lets you group functions as a logical unit for management, deployment, and sharing of resources.

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select `Azure Functions: Create new project...`.
2. Choose a directory location for your project workspace and choose **Select**.

NOTE

These steps were designed to be completed outside of a workspace. For this reason, do not select a project folder that is part of a workspace.

3. Select a language for your function app project. For this tutorial, select **Python**.
4. Select the Python version, (version 3.7.5 is supported by Azure Functions)
5. Select a template for your project's first function. Select **HTTP trigger** to create an HTTP triggered function in the new function app.

6. Provide a function name. In this case, let's use **Concatenator**
7. Select **Function** as the Authorization level. This means that we will provide a [function key](#) to call the function's HTTP endpoint.
8. Select how you would like to open your project. For this step, select **Add to workspace** to create the function app in the current workspace.

Visual Studio Code creates the function app project in a new workspace. This project contains the [host.json](#) and [local.settings.json](#) configuration files, plus any language-specific project files.

A new HTTP triggered function is also created in the **Concatenator** folder of the function app project. Inside it there will be a file called "`__init__.py`", with this content:

```
import logging

import azure.functions as func


def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello {name}!")
    else:
        return func.HttpResponse(
            "Please pass a name on the query string or in the request body",
            status_code=400
        )
```

Now let's modify that code to follow the [custom skill interface](#)). Modify the code with the following content:

```
import logging
import azure.functions as func
import json


def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    try:
        body = json.dumps(req.get_json())
    except ValueError:
        return func.HttpResponse(
            "Invalid body",
            status_code=400
        )

    if body:
        result = compose_response(body)
        return func.HttpResponse(result, mimetype="application/json")
    else:
        return func.HttpResponse(
            "Invalid body",
            status_code=400
        )
```

```

    status_code=400
)

def compose_response(json_data):
    values = json.loads(json_data)['values']

    # Prepare the Output before the loop
    results = {}
    results["values"] = []

    for value in values:
        output_record = transform_value(value)
        if output_record != None:
            results["values"].append(output_record)
    return json.dumps(results, ensure_ascii=False)

## Perform an operation on a record
def transform_value(value):
    try:
        recordId = value['recordId']
    except AssertionError as error:
        return None

    # Validate the inputs
    try:
        assert ('data' in value), "'data' field is required."
        data = value['data']
        assert ('text1' in data), "'text1' field is required in 'data' object."
        assert ('text2' in data), "'text2' field is required in 'data' object."
    except AssertionError as error:
        return (
            {
                "recordId": recordId,
                "errors": [ { "message": "Error:" + error.args[0] } ]
            }
        )

    try:
        concatenated_string = value['data']['text1'] + " " + value['data']['text2']
        # Here you could do something more interesting with the inputs
    except:
        return (
            {
                "recordId": recordId,
                "errors": [ { "message": "Could not complete operation for record." } ]
            }
        )

    return ({
        "recordId": recordId,
        "data": {
            "text": concatenated_string
        }
    })

```

The **transform_value** method performs an operation on a single record. You may modify the method to meet your specific needs. Remember to do any necessary input validation and to return any errors and warnings produced if the operation could not be completed for the record.

Debug your code locally

Visual Studio Code makes it easy to debug the code. Press 'F5' or go to the **Debug** menu and select **Start Debugging**.

You can set any breakpoints on the code by hitting 'F9' on the line of interest.

Once you started debugging, your function will run locally. You can use a tool like Postman or Fiddler to issue the

request to localhost. Note the location of your local endpoint on the Terminal window.

Publish your function

When you're satisfied with the function behavior, you can publish it.

1. In Visual Studio Code, press F1 to open the command palette. In the command palette, search for and select **Deploy to Function App....**
2. Select the Azure Subscription where you would like to deploy your application.
3. Select **+ Create New Function App in Azure**
4. Enter a globally unique name for your function app.
5. Select Python version (Python 3.7.x works for this function).
6. Select a location for the new resource (for example, West US 2).

At this point, the necessary resources will be created in your Azure subscription to host the new Azure Function on Azure. Wait for the deployment to complete. The output window will show you the status of the deployment process.

1. In the [Azure portal](#), navigate to **All Resources** and look for the function you published by its name. If you named it **Concatenator**, select the resource.
2. Click the **</> Get Function URL** button. This will allow you to copy the URL to call the function.

Test the function in Azure

Now that you have the default host key, test your function as follows:

```
POST [Function URL you copied above]
```

Request Body

```
{
  "values": [
    {
      "recordId": "e1",
      "data": {
        "text1": "Hello",
        "text2": "World"
      }
    },
    {
      "recordId": "e2",
      "data": "This is an invalid input"
    }
  ]
}
```

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 concatenate the Title and the Author of the document into a single field which we call merged_title_author. Replace [your-function-url-here] with the URL of your new Azure Function.

```
{  
  "skills": [  
    "[... your existing skills remain here]",  
    {  
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",  
      "description": "Our new search custom skill",  
      "uri": "https://[your-function-url-here]",  
      "context": "/document/merged_content/organizations/*",  
      "inputs": [  
        {  
          "name": "text1",  
          "source": "/document/metadata_title"  
        },  
        {  
          "name": "text2",  
          "source": "/document/metadata_author"  
        },  
      ],  
      "outputs": [  
        {  
          "name": "text",  
          "targetName": "merged_title_author"  
        }  
      ]  
    }  
  ]  
}
```

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](#)

Example: Create a custom skill using the Bing Entity Search API

10/4/2020 • 9 minutes to read • [Edit Online](#)

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 an Azure resource

Start using the Bing Entity Search API by creating one of the following Azure resources.

Bing Entity Search resource

- Available through the Azure portal until you delete the resource.
- Use the free pricing tier to try the service, and upgrade later to a paid tier for production.
- Bing Entity Search is also offered in paid tiers of the [Bing Search v7 resource](#).

Multi-Service resource

- Available through the Azure portal until you delete the resource.
 - Use the same key and endpoint for your applications, across multiple Cognitive Services.
- 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 **Installed**, expand **Visual C# > Cloud**, select **Azure Functions**, type a Name for your project, and select **OK**. 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 **Azure Functions v2 (.NET Core)**. You could also do it with version 1, but the code written below is based on the v2 template.
4. Select the type to be **HTTP Trigger**
5. For Storage Account, you may select **None**, as you won't need any storage for this function.
6. Select **OK** 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:

```
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.cognitive.microsoft.com/bing/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:

```
POST https://localhost:7071/api/EntitySearch
```

Request body

```
{
  "values": [
    {
      "recordId": "e1",
      "data": {
        "name": "Pablo Picasso"
      }
    },
    {
      "recordId": "e2",
      "data": {
        "name": "Microsoft"
      }
    }
  ]
}
```

Response

You should see a response similar to the following example:

```
{
  "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:

```
POST https://[your-entity-search-app-name].azurewebsites.net/api/EntitySearch?code=[enter default host key here]
```

Request Body

```
{
  "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.

```
{
  "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:

```
{
  "@odata.type": "#Microsoft.Skills.Text.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](#)

Example: Create a Form Recognizer custom skill

10/4/2020 • 6 minutes to read • [Edit Online](#)

In this Azure Cognitive Search skillset example, you'll learn how to create a Form Recognizer custom skill using C# and Visual Studio. Form Recognizer analyzes documents and extracts key/value pairs and table data. By wrapping Form Recognizer into the [custom skill interface](#), you can add this capability as a step in an end-to-end enrichment pipeline. The pipeline can then load the documents and do other transformations.

Prerequisites

- [Visual Studio 2019](#) (any edition).
- At least five forms of the same type. You can use sample data provided with this guide.

Create a Form Recognizer resource

Go to the Azure portal and [create a new Form Recognizer resource](#). In the **Create** pane, provide the following information:

Name	A descriptive name for your resource. We recommend using a descriptive name, for example <i>MyNameFormRecognizer</i> .
Subscription	Select the Azure subscription which has been granted access.
Location	The location of your cognitive service instance. Different locations may introduce latency, but have no impact on the runtime availability of your resource.
Pricing tier	The cost of your resource depends on the pricing tier you choose and your usage. For more information, see the API pricing details .
Resource group	The Azure resource group that will contain your resource. You can create a new group or add it to a pre-existing group.

NOTE

Normally when you create a Cognitive Service resource in the Azure portal, you have the option to create a multi-service subscription key (used across multiple cognitive services) or a single-service subscription key (used only with a specific cognitive service). However currently Form Recognizer is not included in the multi-service subscription.

When your Form Recognizer resource finishes deploying, find and select it from the **All resources** list in the portal. Your key and endpoint will be located on the resource's key and endpoint page, under resource management. Save both of these to a temporary location before going forward.

Train your model

You'll need to train a Form Recognizer model with your input forms before you use this skill. Follow the [cURL quickstart](#) to learn how to train a model. You can use the sample forms provided in that quickstart, or you can use your own data. Once the model is trained, copy its ID value to a secure location.

Set up the custom skill

This tutorial uses the [AnalyzeForm](#) project in the [Azure Search Power Skills](#) GitHub repository. Clone this repository to your local machine and navigate to [Vision/AnalyzeForm/](#) to access the project. Then open *AnalyzeForm.csproj* in Visual Studio. This project creates an Azure Function resource that fulfills the [custom skill interface](#) and can be used for Azure Cognitive Search enrichment. It takes form documents as inputs, and it outputs (as text) the key/value pairs that you specify.

First, add project-level environment variables. Locate the **AnalyzeForm** project on the left pane, right-click it and select **Properties**. In the **Properties** window, click the **Debug** tab and then find the **Environment variables** field. Click **Add** to add the following variables:

- `FORMS_RECOGNIZER_ENDPOINT_URL` with the value set to your endpoint URL.
- `FORMS_RECOGNIZER_API_KEY` with the value set to your subscription key.
- `FORMS_RECOGNIZER_MODEL_ID` with the value set to the ID of the model you trained.
- `FORMS_RECOGNIZER_RETRY_DELAY` with the value set to 1000. This value is the time in milliseconds that the program will wait before retrying the query.
- `FORMS_RECOGNIZER_MAX_ATTEMPTS` with the value set to 100. This value is the number of times the program will query the service while attempting to get a successful response.

Next, open *AnalyzeForm.cs* and find the `fieldMappings` variable, which references the *field-mappings.json* file. This file (and the variable that references it) defines the list of keys you want to extract from your forms and a custom label for each key. For example, a value of `{ "Address:", "address" }, { "Invoice For:", "recipient" }` means the script will only save the values for the detected `Address:` and `Invoice For:` fields, and it will label those values with `"address"` and `"recipient"`, respectively.

Finally, note the `contentType` variable. This script runs the given Form Recognizer model on remote documents that are referenced by URL, so the content type is `application/json`. If you want to analyze local files by including their byte streams in the HTTP requests, you'll need to change the `contentType` to the appropriate [MIME type](#) for your file.

Test the function from Visual Studio

After you've edited your project, save it and set the **AnalyzeForm** project as the startup project in Visual Studio (if it isn't set already). Then press **F5** to run the function in your local environment. Use a REST service like [Postman](#) to call the function.

HTTP request

You'll make the following request to call the function.

```
POST https://localhost:7071/api/analyze-form
```

Request body

Start with the request body template below.

```
{
  "values": [
    {
      "recordId": "record1",
      "data": {
        "formUrl": "<your-form-url>",
        "formSasToken": "<your-sas-token>"
      }
    }
  ]
}
```

Here you'll need to provide the URL of a form that has the same type as the forms you trained with. For testing purposes, you can use one of your training forms. If you followed the cURL quickstart, your forms will be located in an Azure blob storage account. Open Azure Storage Explorer, locate a form file, right-click it, and select **Get Shared Access Signature**. The next dialog window will provide a URL and SAS token. Enter these strings in the `"formUrl"` and `"formSasToken"` fields of your request body, respectively.

Name	Access Tier	Access Tier Last Modified	Last Modified	Blob Type	Content Type
Invoice_2.pdf		11/15/2019, 1:24:15 PM	11/15/2019, 1:24:15 PM	Block Blob	application/pdf
Invoice_2.pdf		2/25/2020, 1:42:32 PM	2/25/2020, 1:42:32 PM	Block Blob	application/octet
Invoice_3.pdf		11/19/2019, 11:26:57 AM	11/19/2019, 11:26:57 AM	Block Blob	application/octet
Invoice_3.pdf		11/15/2019, 1:24:15 PM	11/15/2019, 1:24:15 PM	Block Blob	application/pdf
Invoice_3.pdf		2/25/2020, 1:42:30 PM	2/25/2020, 1:42:30 PM	Block Blob	application/octet
Invoice_4.pdf		11/19/2019, 11:28:13 AM	11/19/2019, 11:28:13 AM	Block Blob	application/octet
Invoice_4.pdf		11/15/2019, 1:24:15 PM	11/15/2019, 1:24:15 PM	Block Blob	application/pdf
Invoice_4.pdf		11/19/2019, 11:45:58 AM	11/19/2019, 11:45:58 AM	Block Blob	application/octet
Invoice_4.pdf		11/19/2019, 11:45:49 AM	11/19/2019, 11:45:49 AM	Block Blob	application/octet
Invoice_5.pdf		11/15/2019, 1:24:15 PM	11/15/2019, 1:24:15 PM	Block Blob	application/pdf
Invoice_5.pdf		11/20/2019, 10:04:10 AM	11/20/2019, 10:04:10 AM	Block Blob	application/octet

If you want to analyze a remote document that isn't in Azure blob storage, paste its URL in the `"formUrl"` field and leave the `"formSasToken"` field blank.

NOTE

When the skill is integrated in a skillset, the URL and token will be provided by Cognitive Search.

Response

You should see a response similar to the following example:

```
{  
    "values": [  
        {  
            "recordId": "record1",  
            "data": {  
                "address": "1111 8th st. Bellevue, WA 99501 ",  
                "recipient": "Southridge Video 1060 Main St. Atlanta, GA 65024 "  
            },  
            "errors": null,  
            "warnings": null  
        }  
    ]  
}
```

Publish the function to Azure

When you're satisfied with the function behavior, you can publish it.

1. In the **Solution Explorer** in Visual Studio, 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. 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 new storage account if you don't already have these. When you're finished, select **Create**.
4. After the deployment is complete, notice the Site URL. This URL is the address of your function app in Azure. Save it to a temporary location.
5. In the [Azure portal](#), navigate to the Resource Group, and look for the `AnalyzeForm` Function you published. Under the **Manage** section, you should see Host Keys. Copy the `default` host key and save it to a temporary location.

Connect to your pipeline

To use this skill in a Cognitive Search pipeline, you'll need to add a skill definition to your skillset. The following JSON block is a sample skill definition (you should update the inputs and outputs to reflect your particular scenario and skillset environment). Replace `AzureFunctionEndpointUrl` with your function URL, and replace `AzureFunctionDefaultHostKey` with your host key.

```
{
  "description": "Skillset that invokes the Form Recognizer custom skill",
  "skills": [
    "[... your existing skills go here]",
    {
      "@odata.type": "#Microsoft.Skills.Custom.WebApiSkill",
      "name": "formrecognizer",
      "description": "Extracts fields from a form using a pre-trained form recognition model",
      "uri": "[AzureFunctionEndpointUrl]/api/analyze-form?code=[AzureFunctionDefaultHostKey]",
      "httpMethod": "POST",
      "timeout": "PT30S",
      "context": "/document",
      "batchSize": 1,
      "inputs": [
        {
          "name": "formUrl",
          "source": "/document/metadata_storage_path"
        },
        {
          "name": "formSasToken",
          "source": "/document/metadata_storage_sas_token"
        }
      ],
      "outputs": [
        {
          "name": "address",
          "targetName": "address"
        },
        {
          "name": "recipient",
          "targetName": "recipient"
        }
      ]
    }
  ]
}
```

Next steps

In this guide, you created a custom skill from the Azure Form Recognizer service. To learn more about custom skills, see the following resources.

- [Azure Search Power Skills: a repository of custom skills](#)
- [Add a custom skill to an AI enrichment pipeline](#)
- [Define a skillset](#)
- [Create a skillset \(REST\)](#)
- [Map enriched fields](#)

Tutorial: Build and deploy a custom skill with Azure Machine Learning

10/4/2020 • 5 minutes to read • [Edit Online](#)

In this tutorial, you will use the [hotel reviews dataset](#) (distributed under the Creative Commons license [CC BY-NC-SA 4.0](#)) to create a [custom skill](#) using Azure Machine Learning to extract aspect-based sentiment from the reviews. This allows for the assignment of positive and negative sentiment within the same review to be correctly ascribed to identified entities like staff, room, lobby, or pool.

To train the aspect-based sentiment model in Azure Machine Learning, you will be using the [nlp recipes repository](#). The model will then be deployed as an endpoint on an Azure Kubernetes cluster. Once deployed, the endpoint is added to the enrichment pipeline as an AML skill for use by the Cognitive Search service.

There are two datasets provided. If you wish to train the model yourself, the hotel_reviews_1000.csv file is required. Prefer to skip the training step? Download the hotel_reviews_100.csv.

- Create an Azure Cognitive Search instance
- Create an Azure Machine Learning workspace (the search service and workspace should be in the same subscription)
- Train and deploy a model to an Azure Kubernetes cluster
- Link an AI enrichment pipeline to the deployed model
- Ingest output from deployed model as a custom skill

IMPORTANT

This skill is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). There is currently no .NET SDK support.

Prerequisites

- Azure subscription - get a [free subscription](#).
- [Cognitive Search service](#)
- [Cognitive Services resource](#)
- [Azure Storage account](#)
- [Azure Machine Learning workspace](#)

Setup

- Clone or download the contents of [the sample repository](#).
- Extract contents if the download is a zip file. Make sure the files are read-write.
- While setting up the Azure accounts and services, copy the names and keys to an easily accessed text file. The names and keys will be added to the first cell in the notebook where variables for accessing the Azure services are defined.
- If you are unfamiliar with Azure Machine Learning and its requirements, you will want to review these documents before getting started:
- [Configure a development environment for Azure Machine Learning](#)

- [Create and manage Azure Machine Learning workspaces in the Azure portal](#)
- When configuring the development environment for Azure Machine Learning, consider using the [cloud-based compute instance](#) for speed and ease in getting started.
- Upload the dataset file to a container in the storage account. The larger file is necessary if you wish to perform the training step in the notebook. If you prefer to skip the training step, the smaller file is recommended.

Open notebook and connect to Azure services

1. Put all of the required information for the variables that will allow access to the Azure services inside the first cell and run the cell.
2. Running the second cell will confirm that you have connected to the search service for your subscription.
3. Sections 1.1 - 1.5 will create the search service datastore, skillset, index, and indexer.

At this point you can choose to skip the steps to create the training data set and experiment in Azure Machine Learning and skip directly to registering the two models that are provided in the models folder of the GitHub repo. If you skip these steps, in the notebook you will then skip to section 3.5, Write scoring script. This will save time; the data download and upload steps can take up to 30 minutes to complete.

Creating and training the models

Section 2 has six cells that download the glove embeddings file from the nlp recipes repository. After downloading, the file is then uploaded to the Azure Machine Learning data store. The .zip file is about 2G and it will take some time to perform these tasks. Once uploaded, training data is then extracted and now you are ready to move on to section 3.

Train the aspect based sentiment model and deploy your endpoint

Section 3 of the notebook will train the models that were created in section 2, register those models and deploy them as an endpoint in an Azure Kubernetes cluster. If you are unfamiliar with Azure Kubernetes, it is highly recommended that you review the following articles before attempting to create an inference cluster:

- [Azure Kubernetes service overview](#)
- [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#)
- [Quotas, virtual machine size restrictions, and region availability in Azure Kubernetes Service \(AKS\)](#)

Creating and deploying the inference cluster can take up to 30 minutes. Testing the web service before moving on to the final steps, updating your skillset and running the indexer, is recommended.

Update the skillset

Section 4 in the notebook has four cells that update the skillset and indexer. Alternatively, you can use the portal to select and apply the new skill to the skillset and then run the indexer to update the search service.

In the portal, go to Skillset and select the Skillset Definition (JSON) link. The portal will display the JSON of your skillset that was created in the first cells of the notebook. To the right of the display there is a dropdown menu where you can select the skill definition template. Select the Azure Machine Learning (AML) template. provide the name of the Azure ML workspace and the endpoint for the model deployed to the inference cluster. The template will be updated with the endpoint uri and key.

Skill Definition Templates

Skills

Azure Machine Learning (AML)

Integrate a model built in Azure Machine Learning as a skill.

[Learn more](#)

Workspaces

[dropdown menu]

Endpoints

[dropdown menu]

Show endpoint schema

Template

```
{  
    "@odata.type": "#Microsoft.Skills.Custom.AmlSkill",  
    "uri": "https://contoson7pt12.westus.cloudapp.azure.com:443/api/v1/service/hotel-absa-v2/score",  
    "key": "CfUb0phEqT0fcMIhWkhnh6cbEZxzxgT",  
    "timeout": "PT30S",  
    "degreeOfParallelism": 0,  
    "name": "",  
    "description": "",  
    "context": "",  
    "inputs": [  
        {  
            "name": "",  
            "source": ""  
        }  
    ],  
    "outputs": [  
        {  
            "name": "",  
            "targetName": ""  
        }  
    ]  
}
```

Copy to clipboard 

Copy the skillset template from the window and paste it into the skillset definition on the left. Edit the template to provide the missing values for:

- Name
- Description
- Context
- 'inputs' name and source
- 'outputs' name and targetName

Save the skillset.

After saving the skillset, go to the indexer and select the Indexer Definition (JSON) link. The portal will display the JSON of the indexer that was created in the first cells of the notebook. The output field mappings will need to be updated with additional field mappings to ensure that the indexer can handle and pass them correctly. Save the changes and then select Run.

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

[Review the custom skill web api](#) [Learn more about adding custom skills to the enrichment pipeline](#)

Tips for AI enrichment in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

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 have not done so already, step through the [Tutorial: Learn how to call AI enrichment APIs](#) for practice in applying AI enrichments to a blob data source.

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. The best way to reduce the indexing time is by reducing the number of documents to be indexed.

Start by creating a data source with just a handful of documents/records. Your document sample should be a good representation of the variety of documents that will be indexed.

Run your document sample through the end-to-end pipeline and check that the results meet your needs. Once you are satisfied with the results, you can add more files to your data source.

Tip 2: Make sure your data source credentials are correct

The data source connection is not validated until you define an indexer that uses it. If you see any errors mentioning that the indexer cannot get to the data, make sure that:

- Your connection string is correct. Specially when you are 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 should not depend on this field for indexing purposes.

Add an `enriched` field as part of your index definition for debugging purposes:

Request Body Syntax

```
{
  "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 is not 100%, you can click 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 exceeds 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 [Step 3: Create-an-indexer](#) or see [How to schedule indexers for Azure Cognitive Search](#).

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 datasource 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](#)
- [Specifying data source credentials](#)
- [How to define a skillset](#)
- [How to map enriched fields to an index](#)

Create a knowledge store using REST and Postman

10/4/2020 • 11 minutes to read • [Edit Online](#)

A knowledge store contains output from an Azure Cognitive Search enrichment pipeline for later analysis or other downstream processing. An AI-enriched pipeline accepts image files or unstructured text files, indexes them by using Azure Cognitive Search, applies AI enrichments from Cognitive Services (such as image analysis and natural language processing), and then saves the results to a knowledge store in Azure Storage. You can use tools like Power BI or Storage Explorer in the Azure portal to explore the knowledge store.

In this article, you use the REST API interface to ingest, index, and apply AI enrichments to a set of hotel reviews. The hotel reviews are imported into Azure Blob storage. The results are saved as a knowledge store in Azure Table storage.

After you create the knowledge store, you can learn about how to access the knowledge store by using [Storage Explorer](#) or [Power BI](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

TIP

We recommend [Postman desktop app](#) for this article. The [source code](#) for this article includes a Postman collection containing all of the requests.

Create services and load data

This quickstart 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. Because the data set is so small, you can skip creating or attaching a Cognitive Services resource.

1. [Download HotelReviews_Free.csv](#). This data is hotel review data saved in a CSV file (originates from Kaggle.com) and contains 19 pieces of customer feedback about a single hotel.
2. [Create an Azure storage account](#) or [find an existing account](#) under your current subscription. You'll use Azure storage for both the raw content to be imported, and the knowledge store that is the end result.

Choose the **StorageV2 (general purpose V2)** account type.

3. Open the Blob services pages and create a container named *hotel-reviews*.
4. Click **Upload**.



5. Select the **HotelReviews-Free.csv** file you downloaded in the first step.

Container: hotelreviews

Overview

Access Control (IAM)

Settings

- Access policy
- Properties
- Metadata
- Editor (preview)

Authentication method: Access key (Switch to Azure AD User Account)
Location: hotelreviews-free

NAME	MODIFIED	ACCESS TIER	BLOB TYPE	SIZE	LEASE STATE
HotelReviews_Free.csv	7/29/2019, 11:57:58 AM	Hot (Infe...)	Block blob	8.84 KiB	Available ...

- You are almost done with this resource, but before you leave these pages, use a link on the left navigation pane to open the **Access Keys** page. Get a connection string to retrieve data from Blob storage. A connection string looks similar to the following example:

```
DefaultEndpointsProtocol=https;AccountName=<YOUR-ACCOUNT-NAME>;AccountKey=<YOUR-ACCOUNT-KEY>;EndpointSuffix=core.windows.net
```

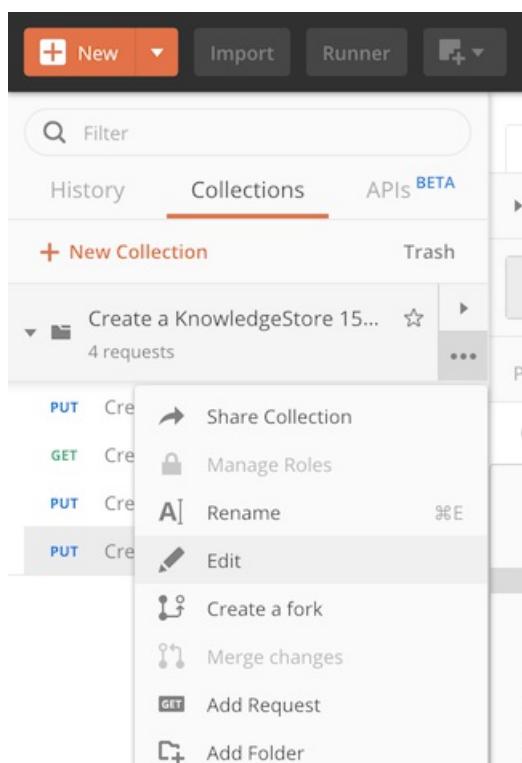
- Still in the portal, switch to Azure Cognitive Search. [Create a new service](#) or [find an existing service](#). You can use a free service for this exercise.

Configure Postman

Install and set up Postman.

Download and install Postman

- Download the [Postman collection source code](#).
- Select **File > Import** to import the source code into Postman.
- Select the **Collections** tab, and then select the ... (ellipsis) button.
- Select **Edit**.



- 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.

To get the value for `admin-key`, go to the Azure Cognitive Search service and select the **Keys** tab. Change `search-service-name` and `storage-account-name` to the values you chose in [Create services](#). Set `storage-connection-string` by using the value on the storage account's **Access Keys** tab. You can leave the defaults for the other values.

EDIT COLLECTION

Name
Create a KnowledgeStore

Description Authorization Pre-request Scripts Tests Variables ●

These variables are specific to this collection and its requests. [Learn more about collection variables](#).

VARIABLE	INITIAL VALUE ⓘ	CURRENT VALUE ⓘ	...	Persist All	Reset All
<input checked="" type="checkbox"/> admin-key	<SEARCH_SERVICE_ADMIN_KEY>	[REDACTED]			
<input checked="" type="checkbox"/> search-service-name	<SEARCH_SERVICE_NAME>	[REDACTED]			
<input checked="" type="checkbox"/> storage-account-name	<STORAGE_ACCOUNT_NAME>	[REDACTED]			
<input checked="" type="checkbox"/> storage-connection-string	<STORAGE_ACCOUNT_CONNEC...	[REDACTED]			
<input checked="" type="checkbox"/> api-version	2019-05-06-Preview	2019-05-06-Preview			
<input checked="" type="checkbox"/> datasource-name	hotel-reviews-ds	hotel-reviews-ds			
<input checked="" type="checkbox"/> indexer-name	hotel-reviews-ixr	hotel-reviews-ixr			
<input checked="" type="checkbox"/> index-name	hotel-reviews-ix	hotel-reviews-ix			
<input checked="" type="checkbox"/> skillset-name	hotel-reviews-ss	hotel-reviews-ss			
<input checked="" type="checkbox"/> storage-container-name	hotel-reviews	hotel-reviews			
Add a new variable					

ⓘ Use variables to reuse values in different places. The current value is used while sending a request and is never synced to Postman's servers. The initial value is auto-updated to reflect the current value. [Change this](#) behaviour from Settings. X

[Learn more about variable values](#)

Cancel Update

VARIABLE	WHERE TO GET IT
<code>admin-key</code>	On the Keys page of the Azure Cognitive Search service.
<code>api-version</code>	Leave as 2020-06-30 .
<code>datasource-name</code>	Leave as hotel-reviews-ds .

VARIABLE	WHERE TO GET IT
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. The URL is <code>https://{{search-service-name}}.search.windows.net</code> .
skillset-name	Leave as hotel-reviews-ss .
storage-account-name	The storage account name.
storage-connection-string	In the storage account, on the Access Keys tab, select key1 > Connection string .
storage-container-name	Leave as hotel-reviews .

Review the request collection in Postman

When you create a knowledge store, you must issue four HTTP requests:

- **PUT request to create the index:** This index holds the data that Azure Cognitive Search uses and returns.
- **POST request to create the datasource:** This datasource connects your Azure Cognitive Search behavior to the data and knowledge store's storage account.
- **PUT request to create the skillset:** The skillset specifies the enrichments that are applied to your data and the structure of the knowledge store.
- **PUT request to create the indexer:** Running the indexer reads the data, applies the skillset, and stores the results. You must run this request last.

The [source code](#) contains a Postman collection that has the four requests. To issue the requests, in Postman, select the tab for the request. Then, add `api-key` and `Content-Type` request headers. Set the value of `api-key` to `{{admin-key}}`. Set the value `Content-type` to `application/json`.

PUT Create Index POST Create Datasource PUT Create the Skillset PUT Create the Indexer

▶ Create Index

PUT https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}?api-version={{api-version}}

Params ● Authorization Headers (10) Body ● Pre-request Script Tests

▼ Headers (2)

KEY	VALUE
api-key	{{{admin-key}}}
Content-Type	application/json

NOTE

You must set `api-key` and `Content-type` headers in all your requests. 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 Azure Cognitive Search index

Create an Azure Cognitive Search index to represent the data that you're interested in searching, filtering, and applying enhancements to. Create the index by issuing a PUT request to

```
https://{{search-service-name}}.search.windows.net/indexes/{{index-name}}?api-version={{api-version}}.
```

Postman replaces symbols that are enclosed in double braces (such as `{{search-service-name}}`, `{{index-name}}`, and `{{api-version}}`) with the values that you set in [Configure Postman](#). If you use a different tool to issue your REST commands, you must substitute those variables yourself.

Set the structure of your Azure Cognitive Search index in the body of the request. In Postman, after you set the `api-key` and `Content-type` headers, go to the **Body** pane of the request. You should see the following JSON. If you don't, select **Raw > JSON (application/json)**, and then paste the following code as the body:

```
{
  "name": "{{index-name}}",
  "fields": [
    { "name": "address", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
    { "name": "categories", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
    { "name": "city", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false },
  ],
  { "name": "country", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "latitude", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "longitude", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "name", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false },
  { "name": "postalCode", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "province", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "reviews_date", "type": "Edm.DateTimeOffset", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "reviews_dateAdded", "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": "metadata_storage_content_type", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "metadata_storage_size", "type": "Edm.Int64", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "metadata_storage_last_modified", "type": "Edm.DateTimeOffset", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "metadata_storage_name", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "metadata_storage_path", "type": "Edm.String", "searchable": false, "filterable": false, "sortable": false, "facetable": false },
  { "name": "Sentiment", "type": "Collection(Edm.Double)", "searchable": false, "filterable": true, "retrievable": true, "sortable": false, "facetable": true },
  { "name": "Language", "type": "Edm.String", "filterable": 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 (the name of the hotel, review content, the date), search metadata, and AI enhancement data (Sentiment, Keyphrases, and Language).

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.

Create the datasource

Next, connect Azure Cognitive Search to the hotel data you stored in Blob storage. To create the datasource, send

a POST request to `https://{{search-service-name}}.search.windows.net/datasources?api-version={{api-version}}`. You must set the `api-key` and `Content-Type` headers as discussed earlier.

In Postman, go to the **Create Datasource** request, and then to the **Body** pane. You should see the following code:

```
{  
    "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 the skillset

The next step is to specify the skillset, which specifies both the enhancements to be applied and the knowledge store where the results will be stored. In Postman, select the **Create the Skillset** tab. This request sends a PUT to `https://{{search-service-name}}.search.windows.net/skillsets/{{skillset-name}}?api-version={{api-version}}`. Set the `api-key` and `Content-type` headers as you did earlier.

There are two large top-level objects: `skills` and `knowledgeStore`. 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.

The `knowledge_store` object connects to the storage account via the `storage-connection-string` Postman variable. `knowledge_store` contains a set of mappings between the enhanced document and tables and columns in the knowledge store.

To generate the skillset, select the **Send** button in Postman to PUT the request:

```
{  
    "name": "{{skillset-name}}",  
    "description": "Skillset to detect language, extract key phrases, and detect sentiment",  
    "skills": [  
        {  
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",  
            "context": "/document/reviews_text", "textSplitMode": "pages", "maximumPageLength": 5000,  
            "inputs": [  
                { "name": "text", "source": "/document/reviews_text" },  
                { "name": "languageCode", "source": "/document/Language" }  
            ],  
            "outputs": [  
                { "name": "textItems", "targetName": "pages" }  
            ]  
        },  
        {  
            "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",  
            "context": "/document/reviews_text/pages/*",  
            "inputs": [  
                { "name": "text", "source": "/document/reviews_text/pages/*" },  
                { "name": "languageCode", "source": "/document/Language" }  
            ],  
            "outputs": [  
                { "name": "score", "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.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": "AzureSearch_DocumentKey", "source": "/document/AzureSearch_DocumentKey" },
            {
                "name": "pages",
                "sourceContext": "/document/reviews_text/pages/*",
                "inputs": [
                    { "name": "SentimentScore", "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" }
            ]
        ],
        "knowledgeStore": {
            "storageConnectionString": "{{storage-connection-string}}",
            "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/*" },
                        { "tableName": "hotelReviewsSentiment", "generatedKeyName": "Sentimentid", "source": "/document/tableprojection/pages/*/sentiment/*" }
                    ],
                    "objects": []
                }
            ]
        }
    }
]

```

```

        },
        {
            "tables": [
                {
                    "tableName": "hotelReviewsInlineDocument", "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_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": "SentimentScore", "source": 
                            "/document/reviews_text/pages/*/Sentiment"}, 
                        { "name": "LanguageCode", "source": "/document/Language"}, 
                        { "name": "Page", "source": "/document/reviews_text/pages/*" }
                    ]
                },
                {
                    "tableName": "hotelReviewsInlineKeyPhrases", "generatedKeyName": "kpidv2",
                    "sourceContext": "/document/reviews_text/pages/*/Keyphrases/*",
                    "inputs": [
                        { "name": "Keyphrases", "source": "/document/reviews_text/pages/*/Keyphrases/*"
                    }
                ],
                "objects": []
            }
        ]
    }
}

```

Create the indexer

The final step is to create the indexer. The indexer reads the data and activates the skillset. In Postman, select the **Create Indexer** request, and then review the body. 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 document that has a header line and comma-separated values. The document key is a unique identifier for the document. Before encoding, the document key is the URL of the source document. Finally, the skillset output values, like language code, sentiment, and key phrases, are mapped to their locations in the document. Although there's a single value for `Language`, `Sentiment` is applied to each element in the array of `pages`. `Keyphrases` is an array that's also applied to each element in the `pages` array.

After you set the `api-key` and `Content-type` headers and confirm that the body of the request is similar to the following source code, select **Send** in Postman. Postman sends a PUT request to

`https://{{search-service-name}}.search.windows.net/indexers/{{indexer-name}}?api-version={{api-version}}`.

Azure Cognitive Search creates and runs the indexer.

```
{
  "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" }
  ]
}
```

Run the indexer

In the Azure portal, go to the Azure Cognitive Search service's [Overview](#) page. Select the **Indexers** tab, and then select **hotels-reviews-ixr**. If the indexer hasn't already run, select **Run**. The indexing task might raise some warnings related to language recognition. The data includes some reviews that are written in languages that aren't yet supported by the cognitive skills.

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 Power BI to explore your enriched data set.

To learn how to explore this knowledge store by using Storage Explorer, see this walkthrough:

[View with Storage Explorer](#)

To learn how to connect this knowledge store to Power BI, see this walkthrough:

[Connect with Power BI](#)

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

View a knowledge store with Storage Explorer

10/4/2020 • 2 minutes to read • [Edit Online](#)

In this article, you'll learn by example how to connect to and explore a knowledge store using Storage Explorer in the Azure portal.

Prerequisites

- Follow the steps in [Create a knowledge store in Azure portal](#) to create the sample knowledge store used in this walkthrough.
 - You will also need the name of the Azure storage account that you used to create the knowledge store, along with its access key from the Azure portal.

View, edit, and query a knowledge store in Storage Explorer

1. In the Azure portal, [open the Storage account](#) that you used to create the knowledge store.
 2. In the storage account's left navigation pane, click **Storage Explorer**.
 3. Expand the **TABLES** list to show a list of Azure table projections that were created when you ran the **Import Data** wizard on your hotel reviews sample data.

Select any table to view the enriched data, including key phrases and sentiment scores.

To change the data type for any table value or to change individual values in your table, click **Edit**. When you change the data type for any column in one table row, it will be applied to all rows.

ADDRESS		CATEGORIES	CITY
ijc3Y7...	Riviera San Nicol 11/a	Hotels	Mableton
ijc3Y7...	Riviera San Nicol 11/a	Hotels	Mableton

Edit Entity

Property Name	Type	Value	
PartitionKey	String	aHR0cHM6Ly9saXNhbGVpYnNhM	
RowKey	String	aHR0cHM6Ly9saXNhbGVpYnNhM	
Timestamp	DateTime	2019-09-09T07:14:17.3880182Z	
AzureSearch_DocumentKey	String	aHR0cHM6Ly9saXNhbGVpYnNhM	
Documentid	String	aHR0cHM6Ly9saXNhbGVpYnNhM	
address	String	Riviera San Nicol 11/a	
categories	String	Hotels	
city	String	Mableton	
country	String	US	
latitude	String	45.421611	
longitude	String	12.376187	

To run queries, click **Query** on the command bar and enter your conditions.

The screenshot shows the Power BI Data view interface. At the top, there is a navigation bar with 'Query' highlighted in red. Below the navigation bar is a table with columns 'ADDRESS', 'CATEGORIES', and 'CITY'. Two rows of data are visible: 'Riviera San Nicol 11/a' under ADDRESS, 'Hotels' under CATEGORIES, and 'Mableton' under CITY. A red arrow points from the 'Query' button in the navigation bar down to the query editor below.

Query Editor:

- And/Or:** And
- Field:** reviews_rating
- Type:** String
- Operator:** >=
- Value:** 4
- And:** reviews_username
- Type:** String
- Operator:** =
- Value:** Julie

Advanced Options: Advanced Options ▾

ALCODE	PROVINCE	REVIEWS_DATE	REVIEWS_DATEADDED	REVIEWS_RATING	REVIEWS_TE
GA		2013-10-27T00:00:00Z	2016-10-24T00:00:25Z	5	We stayed h

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.

Next steps

Connect this knowledge store to Power BI for deeper analysis, or move forward with code, using the REST API and Postman to create a different knowledge store.

[Connect with Power BI](#) [Create a knowledge store in REST](#)

Connect a knowledge store with Power BI

10/4/2020 • 3 minutes to read • [Edit Online](#)

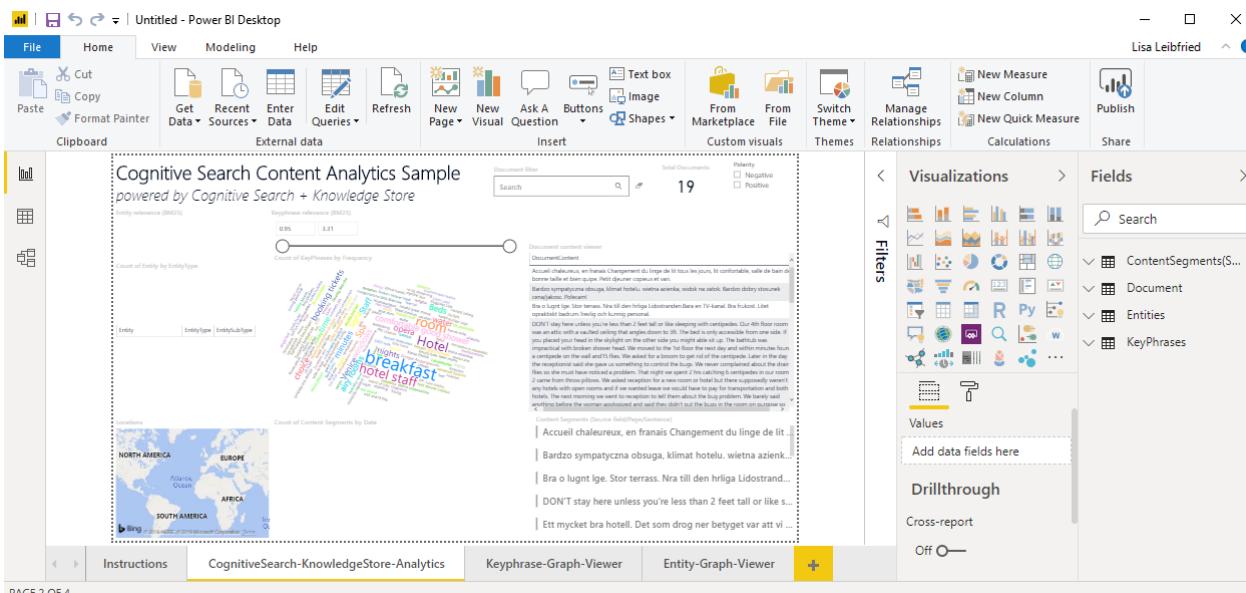
In this article, learn how to connect to and explore a knowledge store using Power Query in the Power BI Desktop app. You can get started faster with templates, or build a custom dashboard from scratch. This brief video below demonstrates how you can enrich your experience with your data by using Azure Cognitive Search in combination with Power BI.

- Follow the steps in [Create a knowledge store in the Azure portal](#) or [Create an Azure Cognitive Search knowledge store by using REST](#) to create the sample knowledge store used in this walkthrough. You will also need the name of the Azure Storage account that you used to create the knowledge store, along with its access key from the Azure portal.
- [Install Power BI Desktop](#)

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

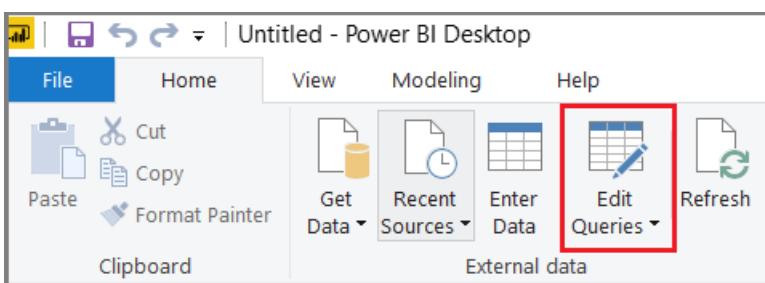
Although the template is downloaded while the wizard is in mid-flight, you'll have to wait until the knowledge store is actually created in Azure Table storage before you can use it.

Connect with Power BI

1. Start Power BI Desktop and click **Get data**.
2. In the **Get Data** window, select **Azure**, and then select **Azure Table Storage**.
3. Click **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.
6. Select the tables containing the hotel reviews data created by the previous walkthroughs.
 - For the portal walkthrough, table names are *hotelReviewsSsDocument*, *hotelReviewsSsEntities*, *hotelReviewsSsKeyPhrases*, and *hotelReviewsSsPages*.
 - For the REST walkthrough, table names are *hotelReviewsDocument*, *hotelReviewsPages*, *hotelReviewsKeyPhrases*, and *hotelReviewsSentiment*.

7. Click **Load**.

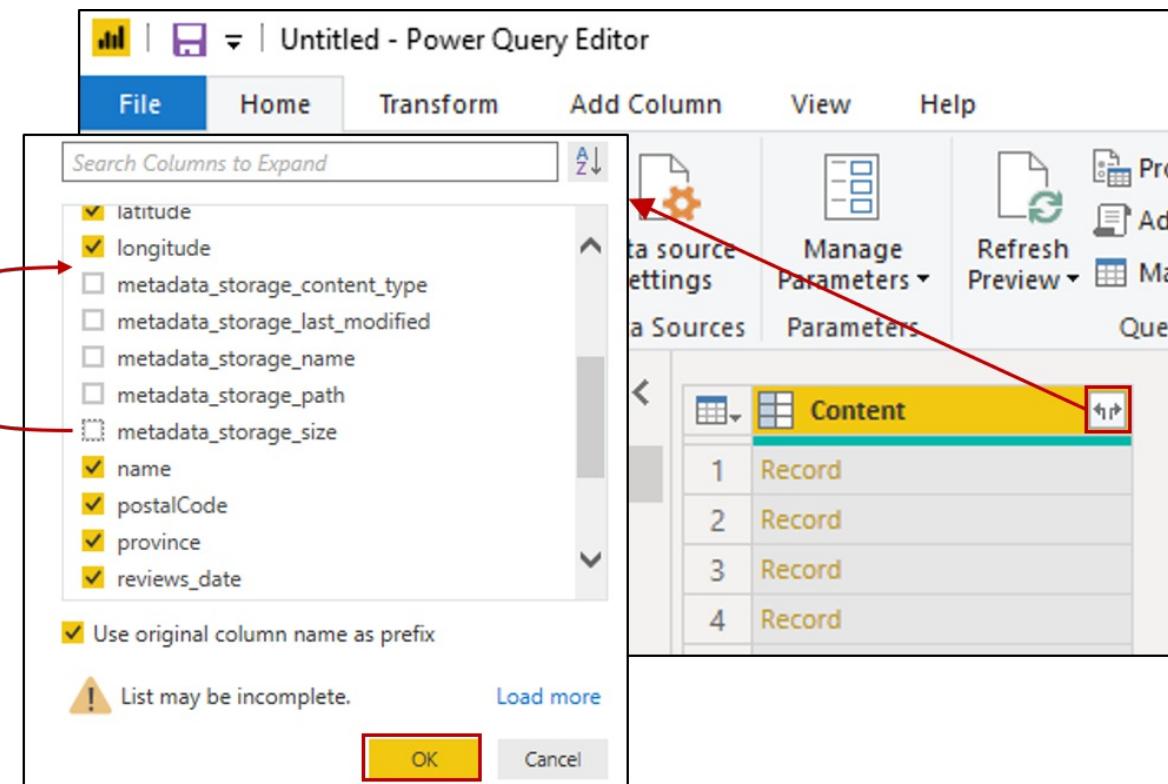
8. On the top ribbon, click **Edit Queries** to open the **Power Query Editor**.



9. Select *hotelReviewsSsDocument*, and then remove the *PartitionKey*, *RowKey*, and *Timestamp* columns.

A screenshot of the Power Query Editor. The 'Queries [3]' pane shows three items: 'hotelReviewsSsDocument', 'hotelReviewsSsKeyPhrases', and 'hotelReviewsSsPages'. The 'hotelReviewsSsDocument' query is selected and previewed. The preview table has four columns: 'PartitionKey', 'RowKey', 'Timestamp', and 'Content'. Three of these columns are crossed out with large red 'X' marks. The 'Content' column is the only one visible in the preview. The 'Transform' ribbon tab is selected, showing various data transformation tools like 'Close & Apply', 'New Source', 'Sources', 'Enter Data', 'Data source settings', 'Parameters', 'Refresh Preview', 'Manage', 'Choose Columns', 'Remove Columns', 'Keep Rows', 'Remove Rows', 'Reduce Rows', 'Sort', 'Split Column', 'Group By', 'Replace Values', and 'Combine'. The 'Query Settings' pane on the right shows the query's properties and applied steps, which currently only lists 'Navigation'.

10. Click the icon with opposing arrows at the upper right side of the table to expand the *Content*. When the list of columns appears, select all columns, and then deselect columns that start with 'metadata'. Click **OK** to show the selected columns.



11. Change the data type for the following columns by clicking the ABC-123 icon at the top left of the column.

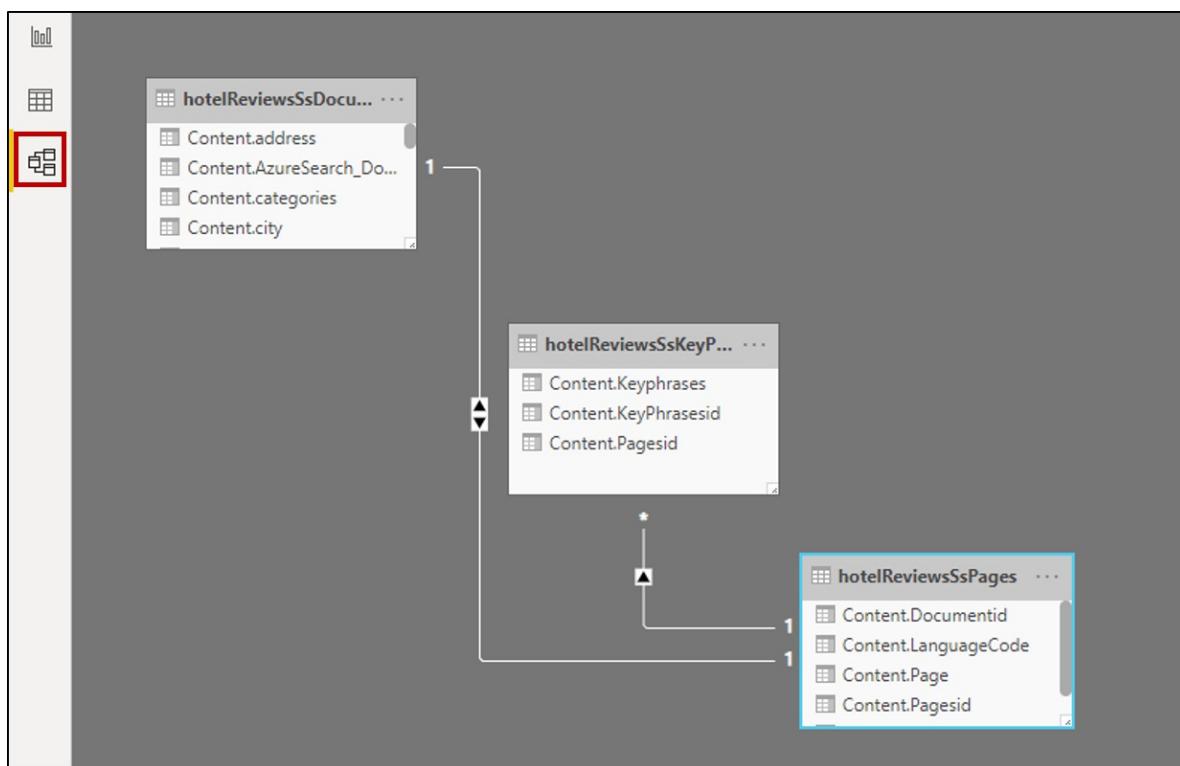
- For *content.latitude* and *Content.longitude*, select **Decimal Number**.
- For *Content.reviews_date* and *Content.reviews_dateAdded*, select **Date/Time**.

This screenshot shows the Power Query Editor with a dropdown menu open for the 'Content.latitude' column. The menu title is 'Content.latitude' and it includes an 'ABC 123' icon. The dropdown contains seven options: '1.2 Decimal Number', '\$ Fixed decimal number', '123 Whole Number', '% Percentage', '5 Date/Time', '6 Date', and '7'. The '1.2 Decimal Number' option is selected. The background shows other columns like 'Content.longitude' and some numerical data.

12. Select *hotelReviewsSsPages*, and then repeat steps 9 and 10 to delete the columns and expand the *Content*.
13. Change the data type for *Content.SentimentScore* to **Decimal Number**.
14. Select *hotelReviewsSsKeyPhrases* and repeat steps 9 and 10 to delete the columns and expand the

Content. There are no data type modifications for this table.

15. On the command bar, click **Close and Apply**.
16. Click on the Model tile on the left navigation pane and validate that Power BI shows relationships between all three tables.



17. 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.
18. Click on the Report tile on the left navigation pane to explore data through visualizations. For text fields, tables and cards are useful visualizations. You can choose fields from each of the three tables to fill in the table or card.

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.

Next steps

To learn how to explore this knowledge store using Storage Explorer, see the following article.

[View with Storage Explorer](#)

How to shape and export enrichments

10/4/2020 • 13 minutes to read • [Edit Online](#)

Projections are the physical expression of enriched documents in a knowledge store. Effective use of enriched documents requires structure. In this article, you'll explore both structure and relationships, learning how to build out projection properties, as well as how to relate data across the projection types created.

To create a projection, the data is shaped using either a [Shaper skill](#) to create a custom object or using the inline shaping syntax within a projection definition.

A data shape contains all the data intended to project, formed as a hierarchy of nodes. This article shows several techniques for shaping data so that it can be projected into physical structures conducive to reporting, analysis, or downstream processing.

The examples presented in this article can be found in this [REST API sample](#), which you can download and run in an HTTP client.

Introduction to projection examples

There are three types of projections:

- Tables
 - Objects
 - Files

Table projections are stored in Azure Table storage. Object and file projections are written to blob storage, where object projections are saved as JSON files, and can contain content from the source document as well as any skill outputs or enrichments. The enrichment pipeline can also extract binaries like images, these binaries are projected as file projections. When a binary object is projected as an object projection, only the metadata associated with it is saved as a JSON blob.

To understand the intersection between data shaping and projections, we'll use the following skillset as the basis for exploring various configurations. This skillset processes raw image and text content. Projections will be defined from the contents of the document and the outputs of the skills, for the desired scenarios.

IMPORTANT

When experimenting with projections, it is useful to [set the indexer cache property](#) to ensure cost control. Editing projections will result in the entire document being enriched again if the indexer cache is not set. When the cache is set and only the projections updated, skillset executions for previously enriched documents do not result in any new Cognitive Services charges.

```
{
  "name": "azureblob-skillset",
  "description": "Skillset created from the portal. skillsetName: azureblob-skillset; contentField: merged_content; enrichmentGranularity: document; knowledgeStoreStorageAccount: confdemo;",
  "skills": [
    {
      "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
      "name": "#1",
      "description": null,
      "context": "/document/merged_content",
      "categories": [
        "Category 1"
      ]
    }
  ]
}
```

```
        "Person",
        "Quantity",
        "Organization",
        "URL",
        "Email",
        "Location",
        "DateTime"
    ],
    "defaultLanguageCode": "en",
    "minimumPrecision": null,
    "includeTypelessEntities": 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"
        },
        {
            "name": "entities",
            "targetName": "entities"
        }
    ]
},
{
    "@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": "DemosCS",
    "key": "<COGNITIVE SERVICES KEY>"
  },
  "knowledgeStore": null
}
```

Using this skillset, with its null `knowledgeStore` as the basis, our first example fills in the `knowledgeStore` object, configured with projections that create tabular data structures we can use in other scenarios.

Projecting to tables

Projecting to tables in Azure Storage is useful for reporting and analysis using tools like Power BI. Power BI can read from tables and discover relationships based on the keys that are generated during projection. If you're trying to build a dashboard, having related data will simplify that task.

Let's build a dashboard to visualize the key phrases extracted from documents as a word cloud. To create the right data structure, add a Shaper skill to the skillset to create a custom shape that has the document-specific details and key phrases. The custom shape will be called `pbiShape` on the `document` root node.

NOTE

Table projections are Azure Storage tables, governed by the storage limits imposed by Azure Storage. For more information, see [table storage limits](#). It is useful to know that 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.

Using a Shaper skill to create a custom shape

Create a custom shape that you can project into table storage. Without a custom shape, a projection can only reference a single node (one projection per output). 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 metadata and identified entities and key phrases. The object is called `pbiShape` and is parented under `/document`.

IMPORTANT

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 are not 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.

```
{
    "@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": "pbShape"
        }
    ]
}
```

Add the above Shaper skill to the skillset.

```

    "name": "azureblob-skillset",
    "description": "A friendly description of the skillset goes here.",
    "skills": [
        {
            "Shaper skill goes here
        }
    ],
    "cognitiveServices": "A key goes here",
    "knowledgeStore": []
}

```

Now that we have all the data needed to project to tables, update the knowledgeStore object with the table definitions. In this example, we have three tables, defined by setting the `tableName`, `source` and `generatedKeyName` properties.

```

"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct Name>;AccountKey=<Acct
Key>",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "pbiDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/pbiShape"
                },
                {
                    "tableName": "pbiKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/pbiShape/keyPhrases/*"
                },
                {
                    "tableName": "pbiEntities",
                    "generatedKeyName": "Entityid",
                    "source": "/document/pbiShape/Entities/*"
                }
            ],
            "objects": [],
            "files": []
        }
    ]
}

```

You can process your work by following these steps:

1. Set the `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 auto-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

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, `pbiShape` is the consolidated shape (or enrichment node). In the projection definition, `pbiShape` 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 will 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/projection 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` is not 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 `pbiDocument` table and the `referenceKeyName` as `DocumentID`. This would result in the column in the `pbiEntities` and `pbiKeyPhrases` tables containing the document ID being named `DocumentID`.

Projecting to objects

Object projections do not have the same limitations as table projections and are better suited for projecting large documents. In this example, the entire document is sent as an object projection. Object projections are limited to a single projection in a container and cannot be sliced.

To define an object projection, use the `objects` array in the 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 our case, `pbiShape`). Inline shaping is useful if you are defining a shape that you do not 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.

```

"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": []
        }
    ]
}

```

Projecting to file

File projections are images that are either extracted from the source document or outputs of enrichment that can be projected out of the enrichment process. File projections, similar to object projections, are implemented as blobs in Azure Storage, and contain the image.

To generate a file projection, use the `files` array in the projection object. This example projects all images extracted from the document to a container called `samplefile`.

```

"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [ ],
            "objects": [ ],
            "files": [
                {
                    "storageContainer": "samplefile",
                    "source": "/document/normalized_images/*"
                }
            ]
        }
    ]
}

```

Projecting to multiple types

A more complex scenario might require you to project content across projection types. For example, if you need to project some data like key phrases and entities to tables, save the OCR results of text and layout text as objects, and then project the images as files.

This example updates the skillset with the following changes:

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.

These changes are reflected in the knowledgeStore definition further down.

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`.

```
{
    "@odata.type": "#Microsoft.Skills.Util.ShaperSkill",
    "name": "ShaperForCross",
    "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).

```
"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"
                        }
                    ]
                },
                {
                    "storageContainer": "crossimages",
                    "generatedKeyName": "crossimages",
                    "source": "/document/crossProjection/images/*/image"
                }
            ]
        }
    ]
}
```

Object projections require a container name for each projection, object projections or file projections cannot 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 is 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 do not want the data related, define the projections in different projection objects. 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.

```
"knowledgeStore" : {
    "storageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<Acct Name>;AccountKey=<Acct Key>;",
    "projections": [
        {
            "tables": [
                {
                    "tableName": "unrelatedDocument",
                    "generatedKeyName": "Documentid",
                    "source": "/document/pbiShape"
                },
                {
                    "tableName": "unrelatedKeyPhrases",
                    "generatedKeyName": "KeyPhraseid",
                    "source": "/document/pbiShape/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": []
        }
    ]
}
```

Common Issues

When defining a projection, there are a few common issues that can cause unanticipated results. Check for these issues if the output in knowledge store isn't what you expect.

- Not shaping string enrichments 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 is not well-formed JSON. So at projection time, make sure to transform the enrichment into a valid JSON object with a name and a value.
- Omitting the `/*` at the end of a source path. If the source of a projection is `/document/pbiShape/keyPhrases`, the key phrases array is projected as a single object/row. Instead, set the source path to `/document/pbiShape/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 do not use the exact case for the selector.

Next steps

The examples in this article demonstrate common patterns on how to create projections. Now that you have a good understanding of the concepts, you are better equipped to build projections for your specific scenario.

As you explore new features, consider incremental enrichment as your next step. Incremental enrichment is based on caching, which lets you reuse any enrichments that are not otherwise affected by a skillset modification. This is especially useful for pipelines that include OCR and image analysis.

[Introduction to incremental enrichment and caching](#)

For an overview on projections, learn more about capabilities like groups and slicing, and how you [define them in a skillset](#)

[Projections in a knowledge store](#)

Query types and composition in Azure Cognitive Search

10/4/2020 • 10 minutes to read • [Edit Online](#)

In Azure Cognitive Search, a query is a full specification of a round-trip operation. On the request, there are parameters that provide execution instructions for the engine, as well as parameters that shape the response coming back. Unspecified (`search=*`), with no match criteria and using null or default parameters, a query executes against all searchable fields as a full text search operation, returning an unscored result set in arbitrary order.

The following example is a representative query constructed in the [REST API](#). This example targets the [hotels demo index](#) and includes common parameters so that you can get an idea of what a query looks like.

```
{  
    "queryType": "simple"  
    "search": "+New York +restaurant",  
    "searchFields": "Description, Address/City, Tags",  
    "select": "HotelId, HotelName, Description, Rating, Address/City, Tags",  
    "top": "10",  
    "count": "true",  
    "orderby": "Rating desc"  
}
```

- `queryType` sets the parser, which is either the [default simple query parser](#) (optimal for full text search), or the [full Lucene query parser](#) used for advanced query constructs like regular expressions, proximity search, fuzzy and wildcard search, to name a few.
- `search` provides the match criteria, usually whole terms or phrases, but often accompanied by boolean operators. Single standalone terms are *term* queries. Quote-enclosed multi-part queries are *phrase* queries. Search can be undefined, as in `search=*`, but with no criteria to match on, the result set is composed of arbitrarily selected documents.
- `searchFields` constrains query execution to specific fields. Any field that is attributed as *searchable* in the index schema is a candidate for this parameter.

Responses are also shaped by the parameters you include in the query:

- `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.

In Azure Cognitive Search, query execution is always against one index, authenticated using an api-key provided in the request. In REST, both are provided in request headers.

How to run this query

Before writing any code, you can use query tools to learn the syntax and experiment with different parameters. The quickest approach is the built-in portal tool, [Search Explorer](#).

If you followed this [quickstart to create the hotels demo index](#), you can paste this query string into the explorer's search bar to run your first query:

```
search+="New York" +restaurant&searchFields=Description, Address/City, Tags&$select=HotelId, HotelName, Description, Rating, Address/City, Tags&$top=10&$orderby=Rating desc&$count=true
```

How query operations are enabled by the index

Index design and query design are tightly coupled in Azure Cognitive Search. An essential fact to know up front is that the *index schema*, with attributes on each field, determines the kind of query you can build.

Index attributes on a field set the allowed operations - whether a field is *searchable* in the index, *retrievable* in results, *sortable*, *filterable*, and so forth. In the example query string, `"$orderby": "Rating"` only works because the Rating field is marked as *sortable* in the index schema.

FIELD NAME	TYPE	RETRIEVABLE	FILTERABLE	SORTABLE	FACTETABLE	SEARCHABLE
HotelId	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
HotelName	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Description_fr	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Category	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tags	Collection(E...)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ParkingIncluded	Edm.Boolean	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
LastRenovationDate	Edm.DateTi...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Rating	Edm.Double	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

The above screenshot is a partial list of index attributes for the hotels sample. You can view the entire index schema in the portal. For more information about index attributes, see [Create Index REST API](#).

NOTE

Some query functionality is enabled index-wide rather than on a per-field basis. These capabilities include: [synonym maps](#), [custom analyzers](#), [suggester constructs \(for autocomplete and suggested queries\)](#), [scoring logic for ranking results](#).

Elements of a query request

Queries are always directed at a single index. You cannot join indexes or create custom or temporary data structures as a query target.

Required elements on a query request include the following components:

- Service endpoint and index documents collection, expressed as a URL containing fixed and user-defined components: `https://<your-service-name>.search.windows.net/indexes/<your-index-name>/docs`
- `api-version` (REST only) is necessary because more than one version of the API is available at all times.
- `api-key`, either a query or admin api-key, authenticates the request to your service.
- `queryType`, either simple or full, which can be omitted if you are using the built-in default simple syntax.
- `search` or `filter` provides the match criteria, which can be unspecified if you want to perform an empty search. Both query types are discussed in terms of the simple parser, but even advanced queries require the

search parameter for passing complex query expressions.

All other search parameters are optional. For the full list of attributes, see [Create Index \(REST\)](#). For a closer look at how parameters are used during processing, see [How full-text search works in Azure Cognitive Search](#).

Choose APIs and tools

The following table lists the APIs and tool-based approaches for submitting queries.

METHODOLOGY	DESCRIPTION
Search explorer (portal)	Provides a search bar and options for index and api-version selections. Results are returned as JSON documents. Recommended for exploration, testing, and validation. Learn more .
Postman or other REST tools	Web testing tools are an excellent choice for formulating REST calls. The REST API supports every possible operation in Azure Cognitive Search. In this article, learn how to set up an HTTP request header and body for sending requests to Azure Cognitive Search.
SearchIndexClient (.NET)	Client that can be used to query an Azure Cognitive Search index. Learn more .
Search Documents (REST API)	GET or POST methods on an index, using query parameters for additional input.

Choose a parser: simple | full

Azure Cognitive Search sits on top of Apache Lucene and gives you a choice between two query parsers for handling typical and specialized queries. Requests using the simple parser are formulated using the [simple query syntax](#), selected as the default for its speed and effectiveness in free form text queries. This syntax supports a number of common search operators including the AND, OR, NOT, phrase, suffix, and precedence operators.

The [full Lucene query syntax](#), enabled when you add `queryType=full` to the request, exposes the widely adopted and expressive query language developed as part of [Apache Lucene](#). Full syntax extends the simple syntax. Any query you write for the simple syntax runs under the full Lucene parser.

The following examples illustrate the point: same query, but with different queryType settings, yield different results. In the first query, the `^3` after `historic` is treated as part of the search term. The top-ranked result for this query is "Marquis Plaza & Suites", which has *ocean* in its description

```
queryType=simple&search=ocean historic^3&searchFields=Description, Tags&$select=HotelId, HotelName, Tags, Description&$count=true
```

The same query using the full Lucene parser interprets `^3` as an in-field term booster. Switching parsers changes the rank, with results containing the term *historic* moving to the top.

```
queryType=full&search=ocean historic^3&searchFields=Description, Tags&$select=HotelId, HotelName, Tags, Description&$count=true
```

Types of queries

Azure Cognitive Search supports a broad range of query types.

QUERY TYPE	USAGE	EXAMPLES AND MORE INFORMATION
Free form text search	Search parameter and either parser	<p>Full text search scans for one or more terms in all <i>searchable</i> fields in your index, and works the way you would expect a search engine like Google or Bing to work. The example in the introduction is full text search.</p> <p>Full text search undergoes lexical analysis using the standard Lucene analyzer (by default) to lower-case all terms, remove stop words like "the". You can override the default with non-English analyzers or specialized language-agnostic analyzers that modify lexical analysis. An 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.</p>
Filtered search	OData filter expression and either parser	<p>Filter queries evaluate a boolean expression over all <i>filterable</i> fields in an index. Unlike search, a filter query matches the exact contents of a field, including case-sensitivity on string fields. Another difference is that filter queries are expressed in OData syntax.</p> <p>Filter expression example</p>
Geo-search	Edm.GeographyPoint type on the field, filter expression, and either parser	<p>Coordinates stored in a field having an Edm.GeographyPoint are used for "find near me" or map-based search controls.</p> <p>Geo-search example</p>
Range search	filter expression and simple parser	<p>In Azure Cognitive Search, range queries are built using the filter parameter.</p> <p>Range filter example</p>
Fielded search	Search parameter and Full parser	<p>Build a composite query expression targeting a single field.</p> <p>Fielded search example</p>
fuzzy search	Search parameter and Full parser	<p>Matches on terms having a similar construction or spelling.</p> <p>Fuzzy search example</p>
proximity search	Search parameter and Full parser	<p>Finds terms that are near each other in a document.</p> <p>Proximity search example</p>
term boosting	Search parameter and Full parser	<p>Ranks a document higher if it contains the boosted term, relative to others that don't.</p> <p>Term boosting example</p>

QUERY TYPE	USAGE	EXAMPLES AND MORE INFORMATION
regular expression search	Search parameter and Full parser	Matches based on the contents of a regular expression. Regular expression example
wildcard or prefix search	Search parameter and Full parser	Matches based on a prefix and tilde (~) or single character (?). Wildcard search example

Manage search results

Query results are streamed as JSON documents in the REST API, although if you use .NET APIs, serialization is built in. You can shape results by setting parameters on the query, selecting specific fields for the response.

Parameters on the query can be used to structure the result set in the following ways:

- Limiting or batching the number of documents in the results (50 by default)
- Selecting fields to include in the results
- Setting a sort order
- Adding hit highlights to draw attention to matching terms in the body of the search results

Tips for unexpected results

Occasionally, the substance and not the structure of results are unexpected. When query outcomes are not what you expect to see, 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.
- Change the query technique if text or lexical analysis is necessary, but the query type precludes linguistic processing. In full text search, text or lexical analysis autocorrects for spelling errors, singular-plural word forms, and even irregular verbs or nouns. For some queries such as fuzzy or wildcard search, lexical analysis is not part of the query parsing pipeline. For some scenarios, regular expressions have been used as a workaround.

Paging results

Azure Cognitive Search makes it easy to implement paging of search results. By using the `top` and `skip` parameters, you can smoothly issue search requests that allow you to receive the total set of search results in manageable, ordered subsets that easily enable good search UI practices. When receiving these smaller subsets of results, you can also receive the count of documents in the total set of search results.

You can learn more about paging search results in the article [How to page search results in Azure Cognitive Search](#).

Ordering results

When receiving results for a search query, you can request that Azure Cognitive Search serves the results ordered by values in a specific field. By default, Azure Cognitive Search orders the search results based on the rank of each document's search score, which is derived from [TF-IDF](#).

If you want Azure Cognitive Search to return your results ordered by a value other than the search score, you can use the `orderby` search parameter. You can specify the value of the `orderby` parameter to include field names and calls to the [geo.distance\(\) function](#) for geospatial values. Each expression can be followed by `asc` to indicate that results are requested in ascending order, and `desc` to indicate that results are requested in descending order. The default ranking ascending order.

Hit highlighting

In Azure Cognitive Search, emphasizing the exact portion of search results that match the search query is made easy by using the `highlight`, `highlightPreTag`, and `highlightPostTag` parameters. You can specify which *searchable* fields should have their matched text emphasized as well as specifying the exact string tags to append to the start and end of the matched text that Azure Cognitive Search returns.

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](#)

Create a simple query in Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

In Azure Cognitive Search, the [simple query syntax](#) invokes the default query parser for executing full text search queries against an index. This parser is fast and handles common scenarios, including full text search, filtered and faceted search, and geo-search.

In this article, we use examples to illustrate the simple syntax, populating the `search=` parameter of a [Search Documents](#) operation.

An alternative query syntax is [Full Lucene](#), supporting more complex query structures, such as fuzzy and wildcard search, which can take additional time to process. For more information and examples demonstrating full syntax, see [Use the full Lucene syntax](#).

Formulate requests in Postman

The following examples leverage a NYC Jobs search index consisting of jobs available based on a dataset provided by the [City of New York OpenData](#) initiative. This data should not be considered current or complete. The index is on a sandbox service provided by Microsoft, which means you do not need an Azure subscription or Azure Cognitive Search to try these queries.

What you do need is Postman or an equivalent tool for issuing HTTP request on GET. For more information, see [Quickstart: Explore Azure Cognitive Search REST API using Postman](#).

Set the request header

1. In the request header, set **Content-Type** to `application/json`.
2. Add an **api-key**, and set it to this string: `252044BE3886FE4A8E3BAA4F595114BB`. This is a query key for the sandbox search service hosting the NYC Jobs index.

After you specify the request header, you can reuse it for all of the queries in this article, swapping out only the `search=` string.

The screenshot shows the Postman interface for a 'NYCjobs' collection. A 'Headers (2)' section is highlighted with a red box. It contains two entries: 'Content-Type: application/json' and 'api-key: 252044BE3886FE4A8E3BAA4F595114BB'. The 'api-key' entry has a description box below it stating: 'This is a query key for a sandbox Azure Search service used for demos'.

Set the request URL

Request is a GET command paired with a URL containing the Azure Cognitive Search endpoint and search string.

The screenshot shows the Postman interface with the URL field containing `https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&search=*`. The entire URL is highlighted with a red box.

URL composition has the following elements:

- `https://azs-playground.search.windows.net/` is a sandbox search service maintained by the Azure Cognitive Search development team.
- `indexes/nycjobs/` is the NYC Jobs index in the indexes collection of that service. Both the service name and

index are required on the request.

- `docs` is the documents collection containing all searchable content. The query api-key provided in the request header only works on read operations targeting the documents collection.
- `api-version=2020-06-30` sets the api-version, which is a required parameter on every request.
- `search=*` is the query string, which in the initial query is null, returning the first 50 results (by default).

Send your first query

As a verification step, paste the following request into GET and click **Send**. Results are returned as verbose JSON documents. Entire documents are returned, which allows you to see all fields and all values.

Paste this URL into a REST client as a validation step and to view document structure.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&search=*
```

The query string, `search=*`, is an unspecified search equivalent to null or empty search. It's not especially useful, but it is the simplest search you can do.

Optionally, you can add `$count=true` to the URL to return a count of the documents matching the search criteria. On an empty search string, this is all the documents in the index (about 2800 in the case of NYC Jobs).

How to invoke simple query parsing

For interactive queries, you don't have to specify anything: simple is the default. In code, if you previously invoked `queryType=full` for full query syntax, you could reset the default with `queryType=simple`.

Example 1: Field-scoped query

This first example is not parser-specific, but we lead with it to introduce the first fundamental query concept: containment. This example scopes query execution and the response to just a few specific fields. Knowing how to structure a readable JSON response is important when your tool is Postman or Search explorer.

For brevity, the query targets only the `business_title` field and specifies only business titles are returned. The syntax is `searchFields` to restrict query execution to just the `business_title` field, and `select` to specify which fields are included in the response.

Partial query string

```
searchFields=business_title&$select=business_title&search=*
```

Here is the same query with multiple fields in a comma-delimited list.

```
search=*&searchFields=business_title, posting_type&$select=business_title, posting_type
```

Full URL

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&searchFields=business_title&$select=business_title&search=*
```

Response for this query should look similar to the following screenshot.

The screenshot shows the Azure Search playground interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (13)', and 'Test Results'. On the right, it displays 'Status: 200 OK' and 'Time: 262 ms'. Below these, there are buttons for 'Pretty', 'Raw', 'Preview', 'JSON' (with a dropdown arrow), and a refresh icon. To the right of the JSON button is a magnifying glass icon and a 'Save Response' button. The main area contains a JSON response with line numbers from 1 to 20. The response is a collection of documents with fields like '@odata.context', '@odata.count', and 'value'. Each document in 'value' has an '@search.score' of 1 and a 'business_title' field containing various job titles.

```

1  {
2    "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",
3    "@odata.count": 2802,
4    "value": [
5      {
6        "@search.score": 1,
7        "business_title": "Business Analyst"
8      },
9      {
10        "@search.score": 1,
11        "business_title": "Limited Alteration Application (LAA) Inspector"
12      },
13      {
14        "@search.score": 1,
15        "business_title": "Deputy Director, Customer Service"
16      },
17      {
18        "@search.score": 1,
19        "business_title": "Borough Safety Specialist"
20      }
]

```

You might have noticed the search score in the response. Uniform scores of 1 occur when there is no rank, either because the search was not full text search, or because no criteria was applied. For null search with no criteria, 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

This example is a bit atypical, but when evaluating search behaviors, you might want to inspect the entire contents of a specific document to understand why it was included or excluded from results. To return a single document in its entirety, use a [Lookup operation](#) to pass in the document ID.

All documents have a unique identifier. To try out the syntax for a lookup query, first return a list of document IDs so that you can find one to use. For NYC Jobs, the identifiers are stored in the `id` field.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&searchFields=id&$select=id&search=*
```

The next example is a lookup query returning a specific document based on `id` "9E1E3AF9-0660-4E00-AF51-9B654925A2D5", which appeared first in the previous response. The following query returns the entire document, not just selected fields.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs/9E1E3AF9-0660-4E00-AF51-9B654925A2D5?api-version=2020-06-30&$count=true&search=*
```

Example 3: Filter queries

[Filter syntax](#) is an OData expression that you can use with `search` or by itself. A standalone filter, without a search parameter, is useful when the filter expression is able to fully qualify documents of interest. Without a query string, there is no lexical or linguistic analysis, no scoring (all scores are 1), and no ranking. Notice the search string is empty.

```

POST /indexes/nycjobs/docs/search?api-version=2020-06-30
{
    "search": "",
    "filter": "salary_frequency eq 'Annual' and salary_range_from gt 90000",
    "select": "job_id, business_title, agency, salary_range_from",
    "count": "true"
}

```

Used together, the 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.

```

1 * {
2     "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(job_id,business_title,
3     ,agency,salary_range_from)",
4     "@odata.count": 96,
5     "value": [
6         {
7             "@search.score": 1,
8             "job_id": "156545",
9             "business_title": "Construction Deputy Director",
10            "agency": "NYC HOUSING AUTHORITY",
11            "salary_range_from": 105000
12        },
13        {
14            "@search.score": 1,
15            "job_id": "175003",
16            "business_title": "Senior Java Developer",
17            "agency": "DEPARTMENT OF CORRECTION",
18            "salary_range_from": 100000
19        },
20        {
21            "@search.score": 1,
22            "job_id": "176839",
23            "business_title": "Internal Monitor",
24            "agency": "ADMIN FOR CHILDREN'S SVCS",
25            "salary_range_from": 110000
26        }
27    ]
28 }

```

If you want to try this out in Postman using GET, you can paste in this string:

```

https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-
30&$count=true&$select=job_id,business_title,agency,salary_range_from&search=&$filter=salary_frequency eq
'Annual' and salary_range_from gt 90000

```

Another powerful way to combine filter and search is through `search.ismatch()` in a filter expression, where you can use a search query within the filter. This filter expression uses a wildcard on `plan` to select `business_title` including the term plan, planner, planning, and so forth.

```

https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-
30&$count=true&$select=job_id,business_title,agency&search=&$filter=search.ismatch('plan*', 'business_title',
'full', 'any')

```

For more information about the function, see [search.ismatch in "Filter examples"](#).

Example 4: Range filters

Range filtering is supported through `$filter` expressions for any data type. The following examples search over numeric and string fields.

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 in Azure Cognitive Search.

The following examples are in POST format for readability (numeric range, followed by text range):

```
POST /indexes/nycjobs/docs/search?api-version=2020-06-30
{
  "search": "",
  "filter": "num_of_positions ge 5 and num_of_positions lt 10",
  "select": "job_id, business_title, num_of_positions, agency",
  "orderby": "agency",
  "count": "true"
}
```

```
1 ▼ {
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(job_id,business_title
, num_of_positions, agency)",
3   "@odata.count": 105,
4   "value": [
5     {
6       "@search.score": 1,
7       "job_id": "191070",
8       "business_title": "Agency Attorney",
9       "num_of_positions": 5,
10      "agency": "ADMIN FOR CHILDREN'S SVCS"
11    },
12    {
13      "@search.score": 1,
14      "job_id": "191070",
15      "business_title": "Agency Attorney",
16      "num_of_positions": 5,
17      "agency": "ADMIN FOR CHILDREN'S SVCS"
18    },
19    {
20      "@search.score": 1,
21      "job_id": "182154",
22      "business_title": "Investigator Level 1",
23      "num_of_positions": 8,
24      "agency": "CIVILIAN COMPLAINT REVIEW BD"
25    },
26  ],
27 }
```

```
POST /indexes/nycjobs/docs/search?api-version=2020-06-30
{
  "search": "",
  "filter": "business_title ge 'A*' and business_title lt 'C**",
  "select": "job_id, business_title, agency",
  "orderby": "business_title",
  "count": "true"
}
```

```
1 ▼ {
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(job_id,business_title
, agency)",
3   "@odata.count": 377,
4   "value": [
5     {
6       "@search.score": 1,
7       "job_id": "187046",
8       "business_title": "ADMINISTRATIVE ACCOUNTANT - LEVEL I",
9       "agency": "DEPARTMENT OF SANITATION"
10    },
11    {
12      "@search.score": 1,
13      "job_id": "187046",
14      "business_title": "ADMINISTRATIVE ACCOUNTANT - LEVEL I",
15      "agency": "DEPARTMENT OF SANITATION"
16    },
17    {
18      "@search.score": 1,
19      "job_id": "123135",
20      "business_title": "ADMINISTRATIVE LAW JUDGE",
21      "agency": "TAX COMMISSION"
22    },
23  ],
24 }
```

You can also try these out in Postman using GET:

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&search=&\$filter=num\_of\_positions%20ge%205%20and%20num\_of\_positions%20lt%2010&\$select=job\_id,%20business\_title,%20num\_of\_positions,%20agency&\$orderby=agency&\$count=true
```

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$search=&$filter=business_title ge 'A*' and business_title lt 'C*&$select=job_id, business_title, agency&$orderby=business_title&$count=true
```

NOTE

Faceting over ranges of values is a common search application requirement. For more information and examples on building filters for facet navigation structures, see "[Filter based on a range](#)" in [How to implement faceted navigation](#).

Example 5: Geo-search

The sample index includes a geo_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 (4) to reduce or enlarge the surface area of the query.

The following example is in POST format for readability:

```
POST /indexes/nycjobs/docs/search?api-version=2020-06-30
{
  "search": "",
  "filter": "geo.distance(geo_location, geography'POINT(-74.11734 40.634384)') le 4",
  "select": "job_id, business_title, work_location",
  "count": "true"
}
```

For more readable results, search results are trimmed to include a job ID, job title, and the work location. The starting coordinates were obtained from a random document in the index (in this case, for a work location on Staten island).

You can also try this out in Postman using GET:

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&count=true&$search=&$select=job_id, business_title, work_location&$filter=geo.distance(geo_location, geography'POINT(-74.11734 40.634384)') le 4
```

Example 6: Search precision

Term queries are single terms, perhaps many of them, that are evaluated independently. Phrase queries are enclosed in quotation marks and evaluated as a verbatim string. Precision of the match is controlled by operators and searchMode.

Example 1: `&search=fire` returns 150 results, where all matches contain the word fire somewhere in the document.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&count=true&$search=fire
```

Example 2: `&search=fire department` returns 2002 results. Matches are returned for documents containing either fire or department.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&count=true&$search=fire department
```

Example 3: `&search="fire department"` returns 82 results. Enclosing the string in quotation marks is a verbatim search on both terms, and matches are found on tokenized terms in the index consisting of the combined terms. This explains why a search like `search=+fire +department` is not equivalent. Both terms are required, but are scanned for independently.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&search="fire department"
```

Example 7: Booleans with searchMode

Simple syntax supports boolean operators in the form of characters (+, -, |). The `searchMode` parameter informs tradeoffs between precision and recall, with `searchMode=any` favoring recall (matching on any criteria qualifies a document for the result set), and `searchMode=all` favoring precision (all criteria must be matched). The default is `searchMode=any`, which 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.

Using the default `searchMode` (any), 2800 documents are returned: those containing the multi-part term "fire department", plus all documents that do not have the term "Metrotech Center".

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&searchMode=any&search="fire department" -"Metrotech Center"
```

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs",  
3   "@odata.count": 2800,  
4   "value": [  
5     {  
6       "@search.score": 0.67712706,  
7       "id": "9896F5B1-1504-43F3-A937-C54465D4B168",  
8       "job_id": "180260",  
9       "agency": "FIRE DEPARTMENT",  
10      "posting_type": "External",  
11      "num_of_positions": 2,  
12      "business_title": "FIELD DESKTOP ASSOCIATE",  
13      "civil_service_title": "COMPUTER ASSOC (SOFTWARE)",  
14      "title_code_no": "13631",  
15      "level": "01",  
16      "salary_range_from": 58721,  
17      "salary_range_to": 81405,  
18      "salary_frequency": "Annual",  
19      "work_location": "5209 5Th Ave., Brooklyn",
```

Changing `searchMode` to `all` enforces a cumulative effect on criteria and returns a smaller result set - 21 documents - consisting of documents containing the entire phrase "fire department", minus those jobs at the Metrotech Center address.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&searchMode=all&search="fire department" -"Metrotech Center"
```

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs",  
3   "@odata.count": 21,  
4   "value": [  
5     {  
6       "@search.score": 0.67712706,  
7       "id": "9896F5B1-1504-43F3-A937-C54465D4B168",  
8       "job_id": "180260",  
9       "agency": "FIRE DEPARTMENT",  
10      "posting_type": "External",  
11      "num_of_positions": 2,  
12      "business_title": "FIELD DESKTOP ASSOCIATE",  
13      "civil_service_title": "COMPUTER ASSOC (SOFTWARE)",  
14      "title_code_no": "13631",
```

Example 8: Structuring results

Several parameters control which fields are in the search results, the number of documents returned in each batch, and sort order. This example resurfaces a few of the previous examples, limiting results to specific fields using the `$select` statement and verbatim search criteria, returning 82 matches

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&$select=job_id,agency,business_title,civil_service_title,work_location,job_description&search="fire department"
```

Appended onto the previous example, you can sort by title. This sort works because `civil_service_title` is *sortable* in the index.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&$select=job_id,agency,business_title,civil_service_title,work_location,job_description&search="fire department"&$orderby=civil_service_title
```

Paging results is implemented using the `$top` parameter, in this case returning the top 5 documents:

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&$select=job_id,agency,business_title,civil_service_title,work_location,job_description&search="fire department"&$orderby=civil_service_title&$top=5&$skip=0
```

To get the next 5, skip the first batch:

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&$select=job_id,agency,business_title,civil_service_title,work_location,job_description&search="fire department"&$orderby=civil_service_title&$top=5&$skip=5
```

Next steps

Try specifying queries in your code. The following links explain how to set up search queries for both .NET and the REST API using the default simple syntax.

- [Query your index using the .NET SDK](#)
- [Query your index using the REST API](#)

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](#)
- [Filter and Orderby syntax](#)

Use the "full" Lucene search syntax (advanced queries in Azure Cognitive Search)

10/4/2020 • 9 minutes to read • [Edit Online](#)

When constructing queries for Azure Cognitive Search, you can replace the default [simple query parser](#) with the more expansive [Lucene Query Parser in Azure Cognitive Search](#) to formulate specialized and advanced query definitions.

The Lucene parser supports complex query constructs, 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 available when using the 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.

Formulate requests in Postman

The following examples leverage a NYC Jobs search index consisting of jobs available based on a dataset provided by the [City of New York OpenData](#) initiative. This data should not be considered current or complete. The index is on a sandbox service provided by Microsoft, which means you do not need an Azure subscription or Azure Cognitive Search to try these queries.

What you do need is Postman or an equivalent tool for issuing HTTP request on GET. For more information, see [Explore with REST clients](#).

Set the request header

1. In the request header, set **Content-Type** to `application/json`.

2. Add an **api-key**, and set it to this string: `252044BE3886FE4A8E3BAA4F595114BB`. This is a query key for the sandbox search service hosting the NYC Jobs index.

After you specify the request header, you can reuse it for all of the queries in this article, swapping out only the `search=` string.

Key	Value	Description
Content-Type	application/json	
api-key	252044BE3886FE4A8E3BAA4F595114BB	This is a query key for a sandbox Azure Search service used for demos

Set the request URL

Request is a GET command paired with a URL containing the Azure Cognitive Search endpoint and search string.



URL composition has the following elements:

- `https://azs-playground.search.windows.net/` is a sandbox search service maintained by the Azure Cognitive Search development team.
- `indexes/nycjobs/` is the NYC Jobs index in the indexes collection of that service. Both the service name and index are required on the request.
- `docs` is the documents collection containing all searchable content. The query api-key provided in the request header only works on read operations targeting the documents collection.
- `api-version=2020-06-30` sets the api-version, which is a required parameter on every request.
- `search=*` is the query string, which in the initial query is null, returning the first 50 results (by default).

Send your first query

As a verification step, paste the following request into GET and click **Send**. Results are returned as verbose JSON documents. Entire documents are returned, which allows you to see all fields and all values.

Paste this URL into a REST client as a validation step and to view document structure.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&$count=true&search=*
```

The query string, `search=*`, is an unspecified search equivalent to null or empty search. It's the simplest search you can do.

Optionally, you can add `$count=true` to the URL to return a count of the documents matching the search criteria. On an empty search string, this is all the documents in the index (about 2800 in the case of NYC Jobs).

How to invoke full Lucene parsing

Add `queryType=full` to invoke the full query syntax, overriding the default simple query syntax.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&search=*
```

All of the examples in this article specify the `queryType=full` search parameter, indicating that the full syntax is handled by the Lucene Query Parser.

Example 1: Query scoped to a list of fields

This first example is not Lucene-specific, but we lead with it to introduce the first fundamental query concept: field scope. This example scopes the entire query and the response to just a few specific fields. Knowing how to structure a readable JSON response is important when your tool is Postman or Search explorer.

For brevity, the query targets only the `business_title` field and specifies only business titles are returned. The `searchFields` parameter restricts query execution to just the `business_title` field, and `select` specifies which fields are included in the response.

Search expression

```
&search=*&searchFields=business_title&$select=business_title
```

Here is the same query with multiple fields in a comma-delimited list.

```
search=*&searchFields=business_title, posting_type&$select=business_title, posting_type
```

The spaces after the commas are optional.

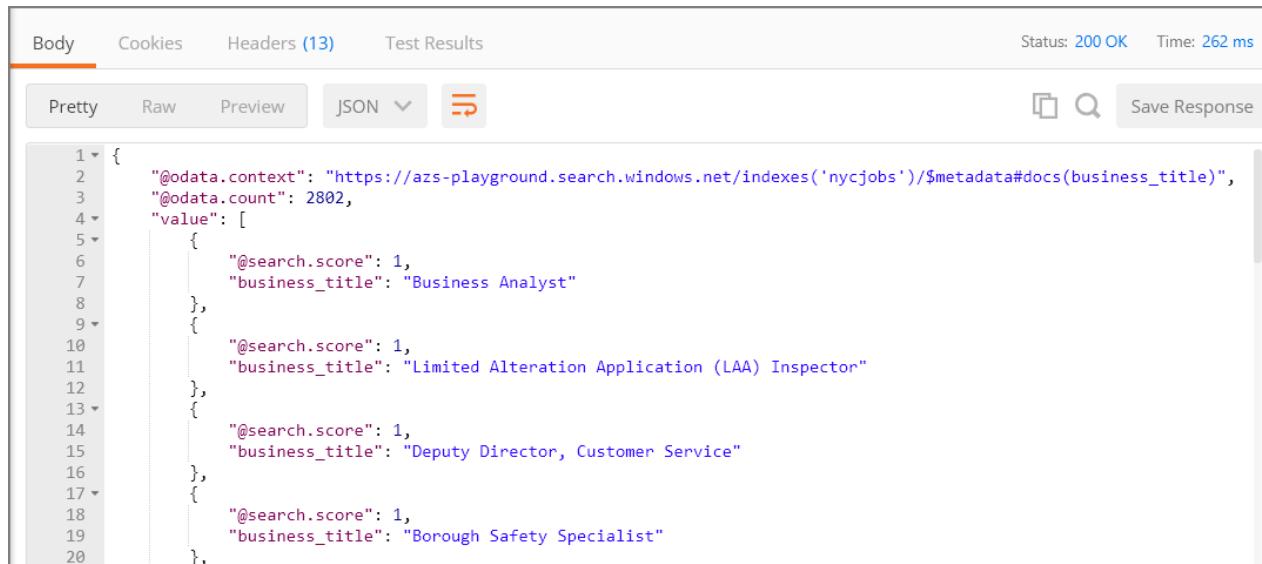
TIP

When using the REST API from your application code, don't forget to URL-encode parameters like `$select` and `searchFields`.

Full URL

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&search=*&searchFields=business_title&$select=business_title
```

Response for this query should look similar to the following screenshot.



```
1  {
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",
3   "@odata.count": 2802,
4   "value": [
5     {
6       "@search.score": 1,
7       "business_title": "Business Analyst"
8     },
9     {
10       "@search.score": 1,
11       "business_title": "Limited Alteration Application (LAA) Inspector"
12     },
13     {
14       "@search.score": 1,
15       "business_title": "Deputy Director, Customer Service"
16     },
17     {
18       "@search.score": 1,
19       "business_title": "Borough Safety Specialist"
20     }
],
```

You might have noticed the search score in the response. Uniform scores of 1 occur when there is no rank, either because the search was not full text search, or because no criteria was applied. For null search with no criteria, rows come back in arbitrary order. When you include actual search criteria, you will see search scores evolve into meaningful values.

Example 2: Fielded search

Full Lucene syntax supports scoping individual search expressions to a specific field. This example searches for business titles with the term senior in them, but not junior.

Search expression

```
$select=business_title&search=business_title:(senior NOT junior)
```

Here is the same query with multiple fields.

```
$select=business_title, posting_type&search=business_title:(senior NOT junior) AND posting_type:external
```

Full URL

[\\$count=true](https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&$select=business_title&$search=business_title:(senior NOT junior))

```
1 {  
2     "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",  
3     "@odata.count": 165,  
4     "value": [  
5         {  
6             "@search.score": 2.6672912,  
7             "business_title": "Senior Estimator"  
8         },  
9         {  
10            "@search.score": 2.465274,  
11            "business_title": "Senior Auditor"  
12        },  
13        {  
14            "@search.score": 2.3447099,  
15            "business_title": "Senior Developer"  
16        }  
17    ]  
18}
```

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:

- business_title:(senior NOT junior)
 - state:(“New York” OR “New Jersey”)
 - business_title:(senior NOT junior) AND posting_type:external

Be sure to put multiple strings 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 `state` field. Also, ensure the operator is capitalized as you see with NOT and AND.

The field specified in `fieldName:searchExpression` must be a searchable field. See [Create Index \(Azure Cognitive Search REST API\)](#) for details on how index attributes are used in field definitions.

NOTE

In the example above, we did not need to use the `searchFields` parameter because each part of the query 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=business_title:(senior NOT junior) AND external&searchFields=posting_type` would match `senior NOT junior` only to the `business_title` field, while it would match "external" with the `posting_type` 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 `business_title` in the `searchFields` parameter.

Example 3: Fuzzy search

Full Lucene syntax also supports fuzzy search, matching on terms that have a similar construction. 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.

Search expression

searchFields=business title&\$select=business title&search=business title:asosiate~

Phrases aren't supported directly but you can specify a fuzzy match on component parts of a phrase.

searchFields=business title&select=business title&search=business title:asosiate~ AND comm~

Full URL

This query searches for jobs with the term "associate" (deliberately misspelled):

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:asosiate~
```

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",  
3   "@odata.count": 85,  
4   "value": [  
5     {  
6       "@search.score": 2.9147263,  
7       "business_title": "Community Associate"  
8     },  
9     {  
10       "@search.score": 2.9147263,  
11       "business_title": "Enforcement Associate"  
12     },  
13     {  
14       "@search.score": 2.821047,  
15       "business_title": "Clerical Associate"  
16     },  
17   ]  
18 }
```

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 incomplete query terms is lowercasing.

Example 4: 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 5 words of each other in a document.

Search expression

```
searchFields=business_title&$select=business_title&search=business_title:%22senior%20analyst%22~1
```

Full URL

In this query, for jobs with the term "senior analyst" where it is separated by no more than one word:

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:%22senior%20analyst%22~1
```

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",  
3   "@odata.count": 18,  
4   "value": [  
5     {  
6       "@search.score": 4.174329,  
7       "business_title": "Senior Analyst"  
8     },  
9     {  
10       "@search.score": 4.174329,  
11       "business_title": "Senior Analyst"  
12     },  
13     {  
14       "@search.score": 4.090341,  
15       "business_title": "Senior Analyst"  
16     },  
17   ]  
18 }
```

Try it again removing the words between the term "senior analyst". Notice that 8 documents are returned for this query as opposed to 10 for the previous query.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:computer%20analyst%22~0
```

Example 5: 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.

Full URLs

In this "before" query, search for jobs with the term *computer analyst* and notice there are no results with both words *computer* and *analyst*, yet *computer* jobs are at the top of the results.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:computer%20analyst
```

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",  
3   "@odata.count": 336,  
4   "value": [  
5     {  
6       "@search.score": 1.6003169,  
7       "business_title": "Computer Associate"  
8     },  
9     {  
10       "@search.score": 1.5311143,  
11       "business_title": "Computer Associate"  
12     },  
13     {  
14       "@search.score": 1.2886862,  
15       "business_title": "Computer Associate (Software)"  
16     },
```

In the "after" query, repeat the search, this time boosting results with the term *analyst* over the term *computer* if both words do not exist.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:computer%20analyst%5e2
```

A more human readable version of the above query is `search=business_title:computer analyst^2`. For a workable query, `^2` is encoded as `%5E2`, which is harder to see.

```
1 {  
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",  
3   "@odata.count": 336,  
4   "value": [  
5     {  
6       "@search.score": 1.249817,  
7       "business_title": "Analyst"  
8     },  
9     {  
10       "@search.score": 1.2403976,  
11       "business_title": "Computer Associate"  
12     },  
13     {  
14       "@search.score": 1.1583551,  
15       "business_title": "Computer Associate"  
16     },
```

Term boosting differs from scoring profiles in that scoring profiles boost certain fields, rather than specific terms. The following example helps illustrate the differences.

Consider a scoring profile that boosts matches in a certain field, such as `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. Furthermore, 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).

When setting the factor level, 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, 0.2).

Example 6: Regex

A regular expression search finds a match based on the contents between forward slashes "/", as documented in the [RegExp class](#).

Search expression

```
searchFields=business_title&$select=business_title&search=business_title:/(Sen|Jun)ior/
```

Full URL

In this query, search for jobs with either the term Senior or Junior: `search=business_title:/(Sen|Jun)ior/`.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:/(Sen|Jun)ior/
```

```
1  {
2      "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",
3      "@odata.count": 183,
4      "value": [
5          {
6              "@search.score": 1,
7              "business_title": "Senior Business Analyst"
8          },
9          {
10             "@search.score": 1,
11             "business_title": "Senior Software Developer"
12         },
13         {
14             "@search.score": 1,
15             "business_title": "Junior Claiming Analyst, Bureau of Budget and Revenue"
16         }
],
```

NOTE

Regex queries are not [analyzed](#). The only transformation performed on incomplete query terms is lowercasing.

Example 7: 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.

Search expression

```
searchFields=business_title&$select=business_title&search=business_title:prog*
```

Full URL

In this query, search for jobs that contain the prefix 'prog' which would include business titles with the terms programming and programmer in it. You cannot use a * or ? symbol as the first character of a search.

```
https://azs-playground.search.windows.net/indexes/nycjobs/docs?api-version=2020-06-30&queryType=full&$count=true&searchFields=business_title&$select=business_title&search=business_title:prog*
```

```
1 {
2   "@odata.context": "https://azs-playground.search.windows.net/indexes('nycjobs')/$metadata#docs(business_title)",
3   "@odata.count": 185,
4   "value": [
5     {
6       "@search.score": 1,
7       "business_title": "Deputy Program Manager"
8     },
9     {
10       "@search.score": 1,
11       "business_title": "MAINFRAME DB2 SYSTEM PROGRAMMER"
12     },
13     {
14       "@search.score": 1,
15       "business_title": "SB1 Data and Program Analyst"
16     },
17     {
18       "@search.score": 1,
19       "business_title": "PROGRAM MANAGER, CAPITAL ACCESS"
```

NOTE

Wildcard queries are not [analyzed](#). The only transformation performed on incomplete query terms is lowercasing.

Next steps

Try specifying the Lucene Query Parser in your code. The following links explain how to set up search queries for both .NET and the REST API. The links use the default simple syntax so you will need to apply what you learned from this article to specify the **queryType**.

- [Query your index using the .NET SDK](#)
- [Query your index using the REST API](#)

Additional syntax reference, query architecture, and examples can be found in the following links:

- [Simple syntax query examples](#)
- [How full text search works in Azure Cognitive Search](#)
- [Simple query syntax](#)
- [Full Lucene query syntax](#)

Partial term search and patterns with special characters (wildcard, regex, patterns)

10/4/2020 • 10 minutes to read • [Edit Online](#)

A *partial term search* refers to queries consisting of term fragments, where instead of a whole term, you might have just the start, 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 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 term search and query strings that include special characters can be problematic if the index doesn't have tokens in the expected format. 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; all of which can cause queries to fail when no match is found. For 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. Likewise, having a complete string that is not tokenized into smaller parts enables pattern matching for "starts with" or "ends with" queries, where the pattern you provide can be evaluated against a term that is not transformed by lexical analysis. Creating an additional field for an intact string, plus using a content-preserving analyzer that emits whole-term tokens, is the solution for both pattern matching and for matching on query strings that include special characters.

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.

What is partial term search in Azure Cognitive 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 partial term or pattern search, and a few other query forms like fuzzy search, analyzers are not 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. Taking a step back, the approach looks like this:

- Define a field to store an intact version of the string (assuming you want analyzed and non-analyzed text at query time)
- Evaluate and choose among the various analyzers that emit tokens at the right level of granularity
- Assign the analyzer to the field
- 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.

Duplicate fields for different scenarios

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.

```
{  
  "name": "featureCode",  
  "type": "Edm.String",  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": null  
},  
{  
  "name": "featureCodeRegex",  
  "type": "Edm.String",  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": "my_custom_analyzer"  
},
```

Choose 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	<p>(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.</p> <p>A recommended combination is the keyword tokenizer with a lower-case token filter. By itself, the predefined keyword analyzer does not 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.</p>

If you are 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.

```
{
  "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 and broken up. Only those queries that match on these tokens will return this document in the results. A query that includes "10-NOR" will fail.

```
{  
  "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:

```
{  
  "text": "SVP10-NOR-00",  
  "analyzer": "keyword"  
}
```

4. Now 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.

```
{  
  "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.

Configure an analyzer

Whether you are evaluating analyzers or moving forward with a specific configuration, you will need to specify the analyzer on the field definition, and possibly configure the analyzer itself if you are not 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 or predefined analyzers can be specified by name on an `analyzer` property of a field definition, with no additional 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 [Predefined analyzers list](#).

```
{  
  "name": "phoneNumber",  
  "type": "Edm.String",  
  "key": false,  
  "retrievable": true,  
  "searchable": true,  
  "analyzer": "whitespace"  
}
```

Use custom analyzers

If you are 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.

```
{  
  "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.

Build and test

Once you have 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 [Choose an analyzer](#). Test some of the strings in your index using a variety of 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 additional token filter processing over existing tokens in your index.

The following example adds an [EdgeNGramTokenFilter](#) to make prefix matches faster. Additional 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 additional 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.

```
{
  "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.

- [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

10/4/2020 • 6 minutes to read • [Edit Online](#)

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 an expansion exercise that produces a match on terms having a similar composition. When a fuzzy search is specified, the 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 is 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, 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, but you can support phrases through AND constructions. For example, "Unviersty~ of~ "Wshington~" would match on "University of Washington".
- 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 additional permutations. This limit is not 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).

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

Analyzers are not used during query processing to create an expansion graph, but that doesn't mean analyzers should be ignored in fuzzy search scenarios. After all, analyzers are used during indexing to create tokens against which matching is done, whether the query is free form, filtered search, or a fuzzy search with a graph as input.

Generally, when assigning analyzers on a per-field basis, the decision to fine-tune the analysis chain is based on the primary use case (a filter or full text search) rather than specialized query forms like fuzzy search. For this reason, there is not a specific analyzer recommendation for fuzzy search.

However, if test queries are not producing the matches you expect, you could try varying the indexing analyzer, setting it to 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 use fuzzy search

Fuzzy queries are constructed using the full Lucene query syntax, invoking the [Lucene query parser](#).

1. Set the full Lucene parser on the query (`queryType=full`).
2. Optionally, scope the request to specific fields, using this parameter (`searchFields=<field1,field2>`).
3. Append the tilde (`~`) operator at the end of the whole term (`search=<string>~`).

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".

In Azure Cognitive Search, besides the term and distance (maximum of 2), there are no additional parameters to set on the query.

NOTE

During query processing, fuzzy queries do not 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.

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 additional 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:

```
search=special~&highlight=Description
```

In the response, because you added hit highlighting, formatting is applied to "special" as the matching term.

```
@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"):

```
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.

```
@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"):

```
search=scal~&highlight=Description
```

Notice that the same response is returned, but now instead of matching on "special", the fuzzy match is on "SQL".

```
@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 \(query parsing architecture\)](#)

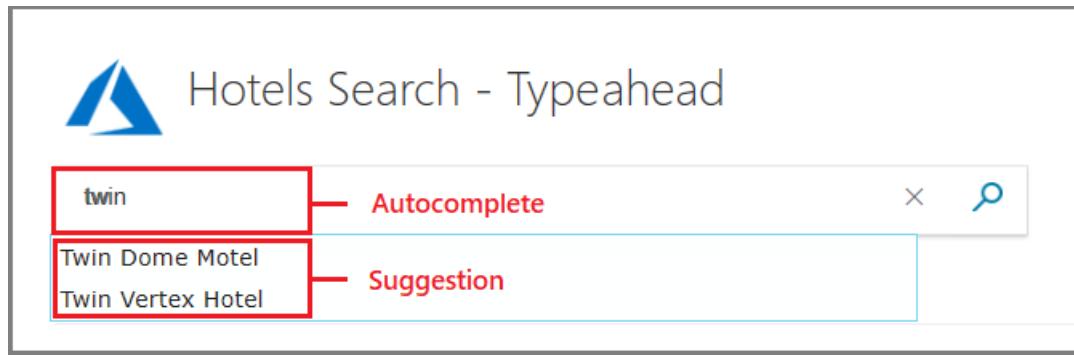
- [Search explorer](#)
- [How to query in .NET](#)
- [How to query in REST](#)

Create a suggester to enable autocomplete and suggested results in a query

10/4/2020 • 6 minutes to read • [Edit Online](#)

In Azure Cognitive Search, "search-as-you-type" is enabled through a **suggester** construct added to a [search index](#). A suggester supports two experiences: *autocomplete*, which completes a partial input for a whole term query, and *suggestions* that invites 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 is an index and query component.

- In the index, add a suggester to an index. You can use the portal, [REST API](#), or [.NET SDK](#). The remainder of this article is focused on creating a suggester.
- In the query request, call 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.

What is a suggester?

A suggester is an internal data structure that supports search-as-you-type behaviors by storing prefixes for matching on partial queries. As with tokenized terms, prefixes are stored in inverted indexes, one for each field specified in a suggester fields collection.

Define a suggester

To create a suggester, add one to an [index schema](#) and [set each property](#). The best time to create a suggester is when you are also defining the field that will use it.

- Use string fields only
- Use the default standard Lucene analyzer (`"analyzer": null`) or a [language analyzer](#) (for example,

```
"analyzer": "en.Microsoft" ) on the field
```

Choose fields

Although a suggester has several properties, it is primarily a collection of string fields for which you are enabling a search-as-you-type experience. There is 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 will 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 would not exist.

When evaluating analyzers, consider using the [Analyze Text API](#) for insight into how terms are tokenized and subsequently prefixed. Once you build an index, you can try various analyzers on a string to view token output.

Fields that use [custom analyzers](#) or [predefined 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.

When to create a suggester

The best time to create a suggester is when you are also creating the field definition itself.

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 will 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](#).

Create using REST

In the REST API, add suggesters through [Create Index](#) or [Update Index](#).

```
{
  "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 [Suggester object](#). `Suggesters` is a collection but it can only take one item.

```
private static void CreateHotelsIndex(SearchServiceClient serviceClient)
{
  var definition = new Index()
  {
    Name = "hotels-sample-index",
    Fields = FieldBuilder.BuildForType<Hotel>(),
    Suggesters = new List<Suggester>() {new Suggester()
    {
      Name = "sg",
      SourceFields = new string[] { "HotelName", "Category" }
    }}
  };
  serviceClient.Indexes.Create(definition);
}
```

Property reference

PROPERTY	DESCRIPTION
<code>name</code>	The name of the suggester.

PROPERTY	DESCRIPTION
<code>searchMode</code>	The strategy used to search for candidate phrases. The only mode currently supported is <code>analyzingInfixMatching</code> , which currently matches on the beginning of a term.
<code>sourceFields</code>	<p>A list of one or more fields that are the source of the content for suggestions. Fields must be of type <code>Edm.String</code> and <code>Collection(Edm.String)</code>. If an analyzer is specified on the field, it must be a named analyzer from this list (not a custom analyzer).</p> <p>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.</p> <p>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.</p>

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](#)
- [SuggestWithHttpMessagesAsync method]
(/dotnet/api/microsoft.azure.search.idocumentoperations.suggestwithhttpmessagesasync?)
- [AutocompleteWithHttpMessagesAsync method](#)

In a search application, client code should leverage 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 is not defined in the index, a call to autocomplete or suggestions will fail.

```
POST /indexes/myboxgames/docs/autocomplete?search&api-version=2020-06-30
{
  "search": "minecraf",
  "suggesterName": "sg"
}
```

Sample code

- [Create your first app in C# \(lesson 3 - Add search-as-you-type\)](#) sample demonstrates a suggester construction, suggested queries, autocomplete, and faceted navigation. This code sample runs on a sandbox Azure Cognitive Search service and uses a pre-loaded Hotels index so all you have to do is press

F5 to run the application. No subscription or sign-in is necessary.

- [DotNetHowToAutocomplete](#) is an older sample containing both C# and Java code. It also demonstrates a suggester construction, suggested queries, autocomplete, and faceted navigation. This code sample uses the hosted [NYCJobs](#) sample data.

Next steps

We recommend the following article to learn more about how requests formulation.

[Add autocomplete and suggestions to client code](#)

Add autocomplete and suggestions to client apps

10/4/2020 • 9 minutes to read • [Edit Online](#)

Search-as-you-type is a common technique for improving the productivity of user-initiated queries. In Azure Cognitive Search, this experience is supported through *autocomplete*, which finishes a term or phrase based on partial input (completing "micro" with "microsoft"). Another form is *suggestions*: a short list of matching documents (returning book titles with an ID so that you can link to a detail page). 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* on the back end.
- 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.

```
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](#)
- [SuggestWithHttpMessagesAsync method](#)
- [AutocompleteWithHttpMessagesAsync 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 [autocompleteMode](#) to determine whether text completion occurs on one or two terms. For Suggestions, the field you choose determines the contents of the response.

For suggestions, you should further refine the response to avoid duplicates or what appears to be unrelated results. To control results, include more parameters on the request. The following parameters apply to both autocomplete and suggestions, but are perhaps more necessary for suggestions, especially when a suggester includes multiple fields.

PARAMETER	USAGE
\$select	If you have multiple sourceFields in a suggester, use \$select to choose which field contributes values (<code>\$select=GameTitle</code>).
searchFields	Constrain the query to specific fields.
\$filter	Apply match criteria on the result set (<code>\$filter=Category eq 'ActionAdventure'</code>).
\$top	Limit the results to a specific number (<code>\$top=5</code>).

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. This article demonstrates two, one for suggestions and another for autocomplete.

- [Autocomplete widget \(jQuery UI\)](#) is used in the Suggestion example. 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](#) is used the Autocomplete example.

We use these libraries to build the search box supporting both suggestions and autocomplete. Inputs collected in the search box are paired with suggestions and autocomplete actions.

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:

```
<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:

```
$(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.

```
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 **microsoft**, **microscope**, and so forth. Highlighting is based on the `HighlightPreTag` and `HighlightPostTag` parameters, defined inline with the `Suggestion` function.

```
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 [DocumentsOperationsExtensions.Suggest 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 `SuggestParameters` 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](#).

```
public ActionResult Suggest(bool highlights, bool fuzzy, string term)
{
    InitSearch();

    // Call suggest API and return results
    SuggestParameters sp = new SuggestParameters()
    {
        Select = HotelName,
        SearchFields = HotelName,
        UseFuzzyMatching = fuzzy,
        Top = 5
    };

    if (highlights)
    {
        sp.HighlightPreTag = "<b>";
        sp.HighlightPostTag = "</b>";
    }

    DocumentSuggestResult resp = _indexClient.Documents.Suggest(term, "sg", sp);

    // Convert the suggest query results to a list that can be displayed in the client.
    List<string> suggestions = resp.Results.Select(x => x.Text).ToList();
    return new JsonResult
    {
        JsonRequestBehavior = JsonRequestBehavior.AllowGet,
        Data = suggestions
    };
}
```

The Suggest function takes two parameters that determine whether hit highlights are returned or fuzzy matching is used in addition to the search term input. The method creates a `SuggestParameters 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`.

```

$(function () {
    // using modified jQuery Autocomplete plugin v1.2.6 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 [DocumentsOperationsExtensions.Autocomplete method](#). As with suggestions, this code block would go in the **HomeController.cs** file.

```

public ActionResult AutoComplete(string term)
{
    InitSearch();
    //Call autocomplete API and return results
    AutocompleteParameters ap = new AutocompleteParameters()
    {
        AutocompleteMode = AutocompleteMode.OneTermWithContext,
        UseFuzzyMatching = false,
        Top = 5
    };
    AutocompleteResult autocompleteResult = _indexClient.Documents.Autocomplete(term, "sg", ap);

    // Convert the Suggest results to a list that can be displayed in the client.
    List<string> autocomplete = autocompleteResult.Results.Select(x => x.Text).ToList();
    return new JsonResult
    {
        JsonRequestBehavior = JsonRequestBehavior.AllowGet,
        Data = 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

Follow these links for end-to-end instructions or code demonstrating both search-as-you-type experiences. Both code examples include hybrid implementations of suggestions and autocomplete together.

- [Tutorial: Create your first app in C# \(lesson 3\)](#)
- [C# code sample: azure-search-dotnet-samples/create-first-app/3-add-typeahead/](#)
- [C# and JavaScript with REST side-by-side code sample](#)

Simple query syntax in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

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 of this flexibility, it is the default for queries in Azure Cognitive Search.

The simple syntax is used for query expressions passed in the `search` parameter of a [Search Documents request](#), not to be confused with the [OData syntax](#) used for the `$filter expressions` parameter of the same Search Documents API. The `search` and `$filter` parameters have different syntax, with their own rules for constructing queries, escaping strings, and so on.

Although the simple parser is based on the [Apache Lucene Simple Query Parser](#) class, the implementation in Azure Cognitive Search excludes fuzzy search. If you need [fuzzy search](#) or other advanced query forms, consider the alternative [full Lucene query syntax](#) instead.

Invoke simple parsing

Simple syntax is the default. Invocation is only necessary if you are resetting the syntax from full to simple. To explicitly set the syntax, use the `queryType` search parameter. Valid values include `queryType=simple` or `queryType=full`, where `simple` is the default, and `full` invokes the [full Lucene query parser](#) for more advanced queries.

Syntax fundamentals

Any text 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 is 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 [NOT operator](#).

Precedence operators (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".

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 is not 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

Please ensure all unsafe and reserved characters are encoded in a URL. For example, '#' is an unsafe character because it is 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. Please see [RFC1738: Uniform Resource Locators \(URL\)](#) for more details.

Unsafe characters are `"`<>#%{}|\\^~[]`. Reserved characters are `;/?:@+=&`.

Querying for special characters

In some circumstances, you may want to search for a special character, like the '♥' emoji or the '€' sign. In those case, make sure that the analyzer you use does not filter those characters out. The standard analyzer ignores many of the special characters so they would not become tokens in your index.

So the first step is to make sure you use an analyzer that will consider those elements tokens. For instance, the "whitespace" analyzer takes into consideration any character sequences separated by whitespaces as tokens, so the "♥" string would be considered a token. Also, an analyzer like the Microsoft English analyzer ("en.microsoft"), would take into consideration 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.

Query size limits

There is a limit to the size of queries that you can send to Azure Cognitive Search. Specifically, you can have at most 1024 clauses (expressions separated by AND, OR, and so on). There is also a limit of approximately 32 KB on the size of any individual term in a query. If your application generates search queries programmatically, we recommend designing it in such a way that it does not generate queries of unbounded size.

Boolean search

You can embed Boolean operators (AND, OR, NOT) in a query string to build a rich set of criteria against which matching documents are found.

AND operator `+`

The AND operator is a plus sign. For example, `wifi + luxury` will search for documents containing both `wifi` and `luxury`.

OR operator |

The OR operator is a vertical bar or pipe character. For example, `wifi | luxury` will search for documents containing either `wifi` or `luxury` or both.

NOT operator -

The NOT operator is a minus sign. For example, `wifi -luxury` will search for documents that have the `wifi` term and/or do not have `luxury`.

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 is no `+` or `|` operator 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, `wifi -luxury` will match documents that either contain the term `wifi` or those that do not contain the term `luxury`.

`searchMode=all` increases the precision of queries by including fewer results, and by default `-` will be interpreted as "AND NOT". For example, `wifi -luxury` will match documents that contain the term `wifi` and do not contain the term "luxury". 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.

Wildcard prefix matching (*, ?)

For "starts with" queries, add a suffix operator as the placeholder for the remainder of a term. Use an asterisk `*` for multiple characters or `?` for single characters. For example, `lingui*` will match on "linguistic" or "linguini", ignoring case.

Similar to filters, a prefix query looks for an exact match. As such, there is 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.

For other wildcard query variants, such as suffix or infix matching against the end or middle of a term, use the [full Lucene syntax for wildcard search](#).

Phrase search "

A term search is a query for 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).

See also

- [How full text search works in Azure Cognitive Search](#)
- [Query examples for simple search](#)
- [Query examples for full Lucene search](#)
- [Search Documents REST API](#)

- [Lucene query syntax](#)
- [OData expression syntax](#)

Lucene query syntax in Azure Cognitive Search

10/4/2020 • 11 minutes to read • [Edit Online](#)

You can write queries against Azure Cognitive Search based on the rich [Lucene Query Parser](#) syntax for specialized query forms: wildcard, fuzzy search, proximity search, regular expressions are a few examples. Much of the Lucene Query Parser syntax is [implemented intact in Azure Cognitive Search](#), with the exception of *range searches* which are constructed in Azure Cognitive Search through `$filter` expressions.

NOTE

The full Lucene syntax is used for query expressions passed in the `search` parameter of the [Search Documents](#) API, not to be confused with the [OData syntax](#) used for the `$filter` parameter of that API. These different syntaxes have their own rules for constructing queries, escaping strings, and so on.

Invoke full parsing

Set the `queryType` search parameter to specify which parser to use. Valid values include `simple|full`, with `simple` as the default, and `full` for Lucene.

Example showing full syntax

The following example finds documents in the index using the Lucene query syntax, evident in the `queryType=full` parameter. This query returns hotels where the category field contains the term "budget" and all searchable fields containing the phrase "recently renovated". Documents containing the phrase "recently renovated" are ranked higher as a result of the term boost value (3).

The `searchMode=all` 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.

```
GET /indexes/hotels/docs?search=category:budget AND \"recently renovated\"^3&searchMode=all&api-version=2020-06-30&querytype=full
```

Alternatively, use POST:

```
POST /indexes/hotels/docs/search?api-version=2020-06-30
{
  "search": "category:budget AND \"recently renovated\"^3",
  "queryType": "full",
  "searchMode": "all"
}
```

For additional examples, see [Lucene query syntax examples for building queries in Azure Cognitive Search](#). For details about specifying the full contingent of query parameters, see [Search Documents \(Azure Cognitive Search REST API\)](#).

NOTE

Azure Cognitive Search also supports [Simple Query Syntax](#), a simple and robust query language that can be used for straightforward keyword search.

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 is not 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

Please ensure all unsafe and reserved characters are encoded in a URL. For example, '#' is an unsafe character because it is 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. Please see [RFC1738: Uniform Resource Locators \(URL\)](#) for more details.

Unsafe characters are " ` < > # % { } | \ ^ ~ [] . Reserved characters are ; / ? : @ = + & .

Query size limits

There is a limit to the size of queries that you can send to Azure Cognitive Search. Specifically, you can have at most 1024 clauses (expressions separated by AND, OR, and so on). There is also a limit of approximately 32 KB on the size of any individual term in a query. If your application generates search queries programmatically, we recommend designing it in such a way that it does not generate queries of unbounded size.

Precedence operators (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".

Boolean search

Always specify text boolean operators (AND, OR, NOT) in all caps.

OR operator `OR` or `||`

The OR operator is a vertical bar or pipe character. For example: `wifi || luxury` will search for 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 || luxury`.

AND operator `AND`, `&&` or `+`

The AND operator is an ampersand or a plus sign. For example: `wifi && luxury` will search for documents containing both "wifi" and "luxury". The plus character (+) is used for required terms. For example, `+wifi +luxury` stipulates that both terms must appear somewhere in the field of a single document.

NOT operator `NOT`, `!` or `-`

The NOT operator is a minus sign. For example, `wifi -luxury` will search for documents that have the `wifi` term and/or do not have `luxury`.

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 is no `+` or `||` operator 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, `wifi -luxury` will match documents that either contain the term `wifi` or those that do not contain the term `luxury`.

`searchMode=all` increases the precision of queries by including fewer results, and by default `-` will be interpreted as "AND NOT". For example, `wifi -luxury` will match documents that contain the term `wifi` and do not contain the term "luxury". 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.

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 5 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 do not 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 are 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 additional 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.

Wildcard search

You can use generally recognized syntax for multiple (`*`) or single (`?`) character wildcard searches. For example, a query expression of `search=alpha*` returns "alphanumeric" or "alphabetical". Note the Lucene query parser supports the use of these symbols with a single term, and not a phrase.

Full Lucene syntax supports prefix, infix, and suffix matching. However, if all you need is prefix matching, you can use the simple syntax (prefix matching is supported in both).

Suffix matching, where `*` or `?` precedes the string (as in `search=.*numeric./`) or infix matching requires full Lucene syntax, as well as the regular expression forward slash `/` delimiters. You cannot use a `*` or `?` symbol as the first character of a term, or within a term, without the `/`.

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. The index will be larger, but queries might execute faster, depending on the pattern construction and the length of strings you are indexing.

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 will 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 are not 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 ([TF-IDF](#)) 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.

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

10/4/2020 • 2 minutes to read • [Edit Online](#)

IMPORTANT

This feature is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). The [REST API version 2020-06-30-Preview](#) provides this feature. There is currently no portal or .NET SDK support.

`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. By default, the contents of all searchable fields are considered, minus any restricted fields that you specified using the `searchFields` parameter. The `moreLikeThis` parameter cannot 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.

You cannot use `MoreLikeThis` on searchable sub-fields in a [complex type](#).

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:

```
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:

```
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:

```
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:

```
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](#)

Filters in Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

A *filter* provides criteria for selecting documents used in an Azure Cognitive Search query. Unfiltered search includes all documents in the index. A filter scopes a search query to a subset of documents. For example, a filter could restrict full text search to just those products having a specific brand or color, at price points above a certain threshold.

Some search experiences impose filter requirements as part of the implementation, but you can use filters anytime you want to constrain search using *value-based* criteria (scoping search to product type "books" for category "non-fiction" published by "Simon & Schuster").

If instead your goal is targeted search on specific data *structures* (scoping search to a customer-reviews field), there are alternative methods, described below.

When to use a filter

Filters are foundational to several search experiences, including "find near me", 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.

Example scenarios include the following:

1. Use a filter to slice your index based on data values in the index. Given a schema with city, housing type, and amenities, you might create a filter to explicitly select documents that satisfy your criteria (in Seattle, condos, waterfront).

Full text search with the same inputs often produces similar results, but a filter is more precise in that it requires an exact match of the filter term against content in your index.

2. Use a filter if the search experience comes with a filter requirement:

- [Faceted navigation](#) uses a filter to pass back the facet category selected by the user.
- Geo-search uses a filter to pass coordinates of the current location in "find near me" apps.
- Security filters pass security identifiers as filter criteria, where a match in the index serves as a proxy for access rights to the document.

3. Use a filter if you want search criteria on a numeric field.

Numeric fields are retrievable in the document and can appear in search results, but they are not searchable (subject to full text search) individually. If you need selection criteria based on numeric data, use a filter.

Alternative methods for reducing scope

If you want a narrowing effect in your search results, filters are not your only choice. These alternatives could be a better fit, depending on your objective:

- `searchFields` query parameter pegs search to specific fields. For example, if your index provides separate fields for English and Spanish descriptions, you can use `searchFields` to target which fields to use for full text search.
- `$select` parameter is used to specify which fields to include in a result set, effectively trimming the response before sending it to the calling application. This parameter does not refine the query or reduce

the document collection, but if a smaller response is your goal, this parameter is an option to consider.

For more information about either parameter, see [Search Documents > Request > Query parameters](#).

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 using a [subset of OData V4 syntax supported in Azure 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 is 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 is 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 does not generate filters of unbounded size.

The following examples represent prototypical filter definitions in several APIs.

```
# Option 1: Use $filter for GET
GET https://[service name].search.windows.net/indexes/hotels/docs?api-version=2020-06-30&search=*&$filter=Rooms/any(room: room/BaseRate lt 150.0)&$select=HotelId, HotelName, Rooms/Description, Rooms/BaseRate

# Option 2: Use filter for POST and pass it in the request body
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"
}
```

```
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 usage 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 is no lexical or linguistic analysis, no scoring, and no ranking. Notice the search string is just an asterisk, which means "match all documents".

```
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.

```
search=walking distance theaters&$filter=Rooms/any(room: room/BaseRate ge 60 and room/BaseRate lt 300) and Address/City eq 'Seattle'&$count=true
```

- 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`.

```
# 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 is 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:

```
$filter=search.ismatchscoring('pool') and Rating ge 4  
  
search=pool&$filter=Rating ge 4
```

Follow up with these articles for comprehensive guidance on specific use cases:

- [Facet filters](#)
- [Language filters](#)
- [Security trimming](#)

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 `Field` object to `true`. You can also do this declaratively by using the `IsFilterable` attribute. In the example below, the attribute is set on the `BaseRate` property of a model class that maps to the index definition.

```
[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. Unlike full-text search, there is 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", `$filter=f eq 'Sunny'` does not match, but `$filter=f eq 'sunny day'` will.

Text strings are case-sensitive. There is no lower-casing of upper-cased words: `$filter=f eq 'Sunny day'` will not find "sunny day".

Approaches for filtering on text

APPROACH	DESCRIPTION	WHEN TO USE
<code>search.in</code>	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 <code>search.in</code> function is designed for speed and is much faster than explicitly comparing the field against each string using <code>eq</code> and <code>or</code> .
<code>search.ismatch</code>	A function that allows you to mix full-text search operations with strictly Boolean filter operations in the same filter expression.	Use <code>search.ismatch</code> (or its scoring equivalent, <code>search.ismatchscoring</code>) when you want multiple search-filter combinations in one request. You can also use it for a <code>contains</code> filter to filter on a partial string within a larger string.
<code>\$filter=field operator string</code>	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 are not `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 will 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 is that full text search itself is not 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:

```
# 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](#)

How to build a facet filter in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

Faceted navigation is used for self-directed filtering on query results in a search app, where your application offers UI controls for scoping search to groups of documents (for example, categories or brands), and Azure Cognitive Search provides the data structure to back the experience. In this article, quickly review the basic steps for creating a faceted navigation structure backing the search experience you want to provide.

- Choose fields for filtering and faceting
- Set attributes on the field
- Build the index and load data
- Add facet filters to a query
- Handle results

Facets are dynamic and returned on a query. Search responses bring with them the facet categories used to navigate the results. If you aren't familiar with facets, the following example is an illustration of a facet navigation structure.

The screenshot shows a search interface with the following sections:

- SEARCH**: Contains a search bar labeled "Search Jobs" with a magnifying glass icon, and a dropdown field "Any distance from" set to "10001".
- FILTER RESULTS**: Contains a section titled "BUSINESS TITLE" with three items:
 - > Auditor (20)
 - > Project Manager (20)
 - > Agency Attorney (14)

A large yellow arrow points from the text "Facets" to the "BUSINESS TITLE" section.

New to faceted navigation and want more detail? See [How to implement faceted navigation in Azure Cognitive Search](#).

Choose fields

Facets can be calculated over single value fields as well as collections. Fields that work best in faceted navigation have low cardinality: a small number of distinct values that repeat throughout documents in your search corpus (for example, a list of colors, countries/regions, or brand names).

Faceting is enabled on a field-by-field basis when you create the index by setting the `facetable` attribute to `true`. You should generally also set the `filterable` attribute to `true` for such fields so that your search application can filter on those fields based on facets that the end user selects.

When creating an index using the REST API, any [field type](#) that could possibly be used in faceted navigation is marked as `facetable` by default:

- `Edm.String`

- `Edm.DateTimeOffset`
- `Edm.Boolean`
- Numeric field types: `Edm.Int32`, `Edm.Int64`, `Edm.Double`
- Collections of the above types (for example, `Collection(Edm.String)` or `Collection(Edm.Double)`)

You cannot 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 is rare that any two sets of coordinates will be equal in a given dataset. As such, facets are not supported for geo-coordinates. You would need a city or region field to facet by location.

Set attributes

Index attributes that control how a field is used are added to individual field definitions in the index. In the following example, fields with low cardinality, useful for faceting, consist of: `category` (hotel, motel, hostel), `tags`, and `rating`. These fields have the `filterable` and `facetable` attributes set explicitly in the following example for illustrative purposes.

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.

```
{
  "name": "hotels",
  "fields": [
    { "name": "hotelId", "type": "Edm.String", "key": true, "searchable": false, "sortable": false,
      "facetable": false },
    { "name": "baseRate", "type": "Edm.Double" },
    { "name": "description", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false },
    { "name": "description_fr", "type": "Edm.String", "filterable": false, "sortable": false, "facetable": false,
      "analyzer": "fr.lucene" },
    { "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": "parkingIncluded", "type": "Edm.Boolean", "filterable": true, "facetable": true, "sortable": false },
    { "name": "smokingAllowed", "type": "Edm.Boolean", "filterable": true, "facetable": true, "sortable": false },
    { "name": "lastRenovationDate", "type": "Edm.DateTimeOffset" },
    { "name": "rating", "type": "Edm.Int32", "filterable": true, "facetable": true },
    { "name": "location", "type": "Edm.GeographyPoint" }
  ]
}
```

NOTE

This index definition is copied from [Create an Azure Cognitive Search index using the REST API](#). It is identical except for superficial differences in the field definitions. The `filterable` and `facetable` attributes are explicitly added on `category`, `tags`, `parkingIncluded`, `smokingAllowed`, and `rating` fields. In practice, `filterable` and `facetable` would be enabled by default on these fields when using the REST API. When using the .NET SDK, these attributes must be enabled explicitly.

Build and load an index

An intermediate (and perhaps obvious) step is that you have to [build and populate the index](#) before formulating a query. We mention this step here for completeness. One way to determine whether the index is available is by checking the indexes list in the [portal](#).

Add facet filters to a query

In application code, construct a query that specifies all parts of a valid query, including search expressions, facets, filters, scoring profiles— anything used to formulate a request. The following example builds a request that creates facet navigation based on the type of accommodation, rating, and other amenities.

```
var sp = new SearchParameters()
{
    ...
    // Add facets
    Facets = new[] { "category", "rating", "parkingIncluded", "smokingAllowed" }.ToList()
};
```

Return filtered results on click events

When the end user clicks on a facet value, the handler for the click event should use a filter expression to realize the user's intent. Given a `category` facet, clicking the category "motel" is implemented with a `$filter` expression that selects accommodations of that type. When a user clicks "motel" to indicate that only motels should be shown, the next query the application sends includes `$filter=category eq 'motel'`.

The following code snippet adds category to the filter if a user selects a value from the category facet.

```
if (!String.IsNullOrEmpty(categoryFacet))
    filter = $"category eq '{categoryFacet}'";
```

If the user clicks on a facet value for a collection field like `tags`, for example the value "pool", your application should use the following filter syntax: `$filter=tags/any(t: t eq 'pool')`

Tips and workarounds

Initialize a page with facets in place

If you want to initialize a page with facets in place, you can send a query as part of page initialization to seed the page with an initial facet structure.

Preserve a facet navigation structure asynchronously of filtered results

One of the challenges with facet 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 facet 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.

See also

- [Filters in Azure Cognitive Search](#)
- [Create Index REST API](#)
- [Search Documents REST API](#)

How to implement faceted navigation in Azure Cognitive Search

10/4/2020 • 22 minutes to read • [Edit Online](#)

Faceted navigation is a filtering mechanism that provides self-directed drilldown navigation in search applications. The term 'faceted navigation' may be unfamiliar, but you've probably used it before. As the following example shows, faceted navigation is nothing more than the categories used to filter results.

The screenshot displays the Azure Search Job Portal Demo interface. At the top, there's a header with the Azure Search logo and the text "AVAILABLE JOBS (2802 jobs)". Below the header, the main content area is divided into several sections:

- SEARCH:** Includes a search bar labeled "Search Jobs" and a dropdown for "Any distance from" set to "10001".
- FILTER RESULTS:** Contains two expandable sections: "BUSINESS TITLE" and "LOCATION". The "BUSINESS TITLE" section is highlighted with a red border and lists categories like Auditor, Project Manager, Agency Attorney, etc. The "LOCATION" section lists Internal (1469) and External (1333).
- MAP:** A map of New York City showing job locations marked with orange dots. A red callout box labeled "FACETS" points to the facet navigation area on the left.
- RESULTS COUNT:** "2802 AVAILABLE JOBS"
- RELEVANCE FILTER:** A dropdown menu set to "Relevance".
- PAGINATION:** A page number indicator showing pages 1 through 5.
- LISTING:** A job listing for "Java Developer - Featured Job" at Metro Tech 4Th Flr, Rm 418, posted on Jan 4. It shows a salary range of \$81,290 to \$100,000 Annual. A snippet of the job description mentions DoITT providing services to New York City residents, businesses, employees, and visitors.

Faceted navigation is an alternative entry point to search. It offers a convenient alternative to typing complex search expressions by hand. 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. In online retail applications, faceted navigation is often built over brands, departments (kid's shoes), size, price, popularity, and ratings.

Implementing faceted navigation differs across search technologies. In Azure Cognitive Search, faceted navigation is built at query time, using fields that you previously attributed in your schema.

- In the queries that your application builds, a query must send *facet query parameters* to get the available facet filter values for that document result set.
- To actually trim the document result set, the application must also apply a `$filter` expression.

In your application development, writing code that constructs queries constitutes the bulk of the work. Many of the application behaviors that you would expect from faceted navigation are provided by the service, including built-in support for defining ranges and getting counts for facet results. The service also includes sensible defaults that help you avoid unwieldy navigation structures.

Sample code and demo

This article uses a job search portal as an example. The example is implemented as an ASP.NET MVC application.

- See and test the working demo online at [Azure Cognitive Search Job Portal Demo](#).
- Download the code from the [Azure-Samples repo on GitHub](#).

Get started

If you're new to search development, the best way to think of faceted navigation is that it shows the possibilities for self-directed search. It's a type of drill-down search experience, based on predefined filters, used for quickly narrowing down search results through point-and-click actions.

Interaction model

The search experience for faceted navigation is iterative, so let's start by understanding it as a sequence of queries that unfold in response to user actions.

The starting point is an application page that provides faceted navigation, typically placed on the periphery. Faceted navigation is often a tree structure, with checkboxes for each value, or clickable text.

1. A query sent to Azure Cognitive Search specifies the faceted navigation structure via one or more facet query parameters. For instance, the query might include `facet=Rating`, perhaps with a `:values` or `:sort` option to further refine the presentation.
2. The presentation layer renders a search page that provides faceted navigation, using the facets specified on the request.
3. Given a faceted navigation structure that includes Rating, you click "4" to indicate that only products with a rating of 4 or higher should be shown.
4. In response, the application sends a query that includes `$filter=Rating ge 4`
5. The presentation layer updates the page, showing a reduced result set, containing just those items that satisfy the new criteria (in this case, products rated 4 and up).

A facet is a query parameter, but do not confuse it with query input. It is never used as selection criteria in a query. Instead, think of facet query parameters as inputs to the navigation structure that comes back in the response. For each facet query parameter you provide, Azure Cognitive Search evaluates how many documents are in the partial results for each facet value.

Notice the `$filter` in step 4. The filter is an important aspect of faceted navigation. Although facets and filters are independent in the API, you need both to deliver the experience you intend.

App design pattern

In application code, the pattern is to use facet query parameters to return the faceted navigation structure along with facet results, plus a `$filter` expression. The filter expression handles the click event on the facet value. Think of the `$filter` expression as the code behind the actual trimming of search results returned to the presentation layer. Given a Colors facet, clicking the color Red is implemented through a `$filter` expression that selects only those items that have a color of red.

Query basics

In Azure Cognitive Search, a request is specified through one or more query parameters (see [Search Documents](#) for a description of each one). None of the query parameters are required, but you must have at least one in order for a query to be valid.

Precision, understood as the ability to filter out irrelevant hits, is achieved through one or both of these expressions:

- `search=`

The value of this parameter constitutes the search expression. It might be a single piece of text, or a complex search expression that includes multiple terms and operators. On the server, a search expression is used for full-text search, querying searchable fields in the index for matching terms, returning results in rank order. If you set `search` to null, query execution is over the entire index (that is, `search=*`). In this case, other elements of the query, such as a `$filter` or scoring profile, are the primary factors affecting which documents are returned (`$filter`) and in what order (`scoringProfile` or `$orderby`).

- **\$filter=**

A filter is a powerful mechanism for limiting the size of search results based on the values of specific document attributes. A `$filter` is evaluated first, followed by faceting logic that generates the available values and corresponding counts for each value.

Complex search expressions decrease the performance of the query. Where possible, use well-constructed filter expressions to increase precision and improve query performance.

To better understand how a filter adds more precision, compare a complex search expression to one that includes a filter expression:

- `GET /indexes/hotel/docs?search=lodging budget +Seattle -motel +parking`
- `GET /indexes/hotel/docs?search=lodging&$filter=City eq 'Seattle' and Parking and Type ne 'motel'`

Both queries are valid, but the second is superior if you're looking for non-motels with parking in Seattle.

- The first query relies on those specific words being mentioned or not mentioned in string fields like Name, Description, and any other field containing searchable data.
- The second query looks for precise matches on structured data and is likely to be much more accurate.

In applications that include faceted navigation, make sure that each user action over a faceted navigation structure is accompanied by a narrowing of search results. To narrow results, use a filter expression.

Build a faceted navigation app

You implement faceted navigation with Azure Cognitive Search in your application code that builds the search request. The faceted navigation relies on elements in your schema that you defined previously.

Predefined in your search index is the `Facetable [true|false]` index attribute, set on selected fields to enable or disable their use in a faceted navigation structure. Without `"Facetable" = true`, a field cannot be used in facet navigation.

The presentation layer in your code provides the user experience. It should list the constituent parts of the faceted navigation, such as the label, values, check boxes, and the count. The Azure Cognitive Search REST API is platform agnostic, so use whatever language and platform you want. The important thing is to include UI elements that support incremental refresh, with updated UI state as each additional facet is selected.

At query time, your application code creates a request that includes `facet=[string]`, a request parameter that provides the field to facet by. A query can have multiple facets, such as `&facet=color&facet=category&facet=rating`, each one separated by an ampersand (&) character.

Application code must also construct a `$filter` expression to handle the click events in faceted navigation. A `$filter` reduces the search results, using the facet value as filter criteria.

Azure Cognitive Search returns the search results, based on one or more terms that you enter, along with updates to the faceted navigation structure. In Azure Cognitive Search, faceted navigation is a single-level construction, with facet values, and counts of how many results are found for each one.

In the following sections, we take a closer look at how to build each part.

Build the index

Faceting is enabled on a field-by-field basis in the index, via this index attribute: `"Facetable": true`.

All field types that could possibly be used in faceted navigation are `Facetable` by default. Such field types include `Edm.String`, `Edm.DateTimeOffset`, and all the numeric field types (essentially, all field types are facetable except `Edm.GeographyPoint`, which can't be used in faceted navigation).

When building an index, a best practice for faceted navigation is to explicitly turn facetting off for fields that should never be used as a facet. In particular, string fields for singleton values, such as an ID or product name, should be set to `"Facetable": false` to prevent their accidental (and ineffective) use in faceted navigation. Turning facetting off where you don't need it helps keep the size of the index small, and typically improves performance.

Following is part of the schema for the Job Portal Demo sample app, trimmed of some attributes to reduce the size:

```
{  
  ...  
  "name": "nycjobs",  
  "fields": [  
    { "name": "id", "type": "Edm.String", "searchable": false, "filterable": false, ... "facetable": false, ... },  
    { "name": "job_id", "type": "Edm.String", "searchable": false, "filterable": false, ... "facetable": false, ... },  
    { "name": "agency", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "posting_type", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "num_of_positions", "type": "Edm.Int32", "searchable": false, "filterable": true, ... "facetable": true, ... },  
    { "name": "business_title", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "civil_service_title", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "title_code_no", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "level", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "salary_range_from", "type": "Edm.Int32", "searchable": false, "filterable": true, ... "facetable": true, ... },  
    { "name": "salary_range_to", "type": "Edm.Int32", "searchable": false, "filterable": true, ... "facetable": true, ... },  
    { "name": "salary_frequency", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    { "name": "work_location", "type": "Edm.String", "searchable": true, "filterable": true, ... "facetable": true, ... },  
    ...  
    { "name": "geo_location", "type": "Edm.GeographyPoint", "searchable": false, "filterable": true, ... "facetable": false, ... },  
    { "name": "tags", "type": "Collection(Edm.String)", "searchable": true, "filterable": true, ... "facetable": true, ... }  
  ],  
  ...  
}
```

As you can see in the sample schema, `Facetable` is turned off for string fields that shouldn't be used as facets, such as ID values. Turning facetting off where you don't need it helps keep the size of the index small, and typically improves performance.

TIP

As a best practice, include the full set of index attributes for each field. Although `Facetable` is on by default for almost all fields, purposely setting each attribute can help you think through the implications of each schema decision.

Check the data

The quality of your data has a direct effect on whether the faceted navigation structure materializes as you expect it to. It also affects the ease of constructing filters to reduce the result set.

If you want to facet by Brand or Price, each document should contain values for *BrandName* and *ProductPrice* that are valid, consistent, and productive as a filter option.

Here are a few reminders of what to scrub for:

- For every field that you want to facet by, ask yourself whether it contains values that are suitable as filters in self-directed search. The values should be short, descriptive, and sufficiently distinctive to offer a clear choice between competing options.
- Misspellings or nearly matching values. If you facet on Color, and field values include Orange and Ornage (a misspelling), a facet based on the Color field would pick up both.
- Mixed case text can also wreak havoc in faceted navigation, with orange and Orange appearing as two different values.
- Single and plural versions of the same value can result in a separate facet for each.

As you can imagine, diligence in preparing the data is an essential aspect of effective faceted navigation.

Build the UI

Working back from the presentation layer can help you uncover requirements that might be missed otherwise, and understand which capabilities are essential to the search experience.

In terms of faceted navigation, your web or application page displays the faceted navigation structure, detects user input on the page, and inserts the changed elements.

For web applications, AJAX is commonly used in the presentation layer because it allows you to refresh incremental changes. You could also use ASP.NET MVC or any other visualization platform that can connect to an Azure Cognitive Search service over HTTP. The sample application referenced throughout this article -- the [Azure Cognitive Search Job Portal Demo](#) – happens to be an ASP.NET MVC application.

In the sample, faceted navigation is built into the search results page. The following example, taken from the `index.cshtml` file of the 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.

```

<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>

```

The following code snippet from the `index.cshtml` page 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.

```

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);
}

```

TIP

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. For example, in the Job Search Portal sample application, you can click the `[x]` after a selected facet to clear the facet.

Build the query

The code that you write for building queries should specify all parts of a valid query, including search expressions, facets, filters, scoring profiles— anything used to formulate a request. In this section, we explore where facets fit into a query, and how filters are used with facets to deliver a reduced result set.

Notice that facets are integral in this sample application. The search experience in the Job Portal Demo is designed around faceted navigation and filters. The prominent placement of faceted navigation on the page demonstrates its importance.

An example is often a good place to begin. The following example, taken from the `JobsSearch.cs` file, builds a request that creates facet navigation based on Business Title, Location, Posting Type, and Minimum Salary.

```

SearchParameters sp = new SearchParameters()
{
    ...
    // Add facets
    Facets = new List<String>() { "business_title", "posting_type", "level", "salary_range_from,interval:50000"
},
};

```

A facet query parameter is set to a field and depending on the data type, can be further parameterized by comma-delimited list that includes `count:<integer>`, `sort:<>`, `interval:<integer>`, and `values:<list>`. A values list is supported for numeric data when setting up ranges. See [Search Documents \(Azure Cognitive Search API\)](#) for usage details.

Along with facets, the request formulated by your application should also build filters to narrow down the set of candidate documents based on a facet value selection. For a bike store, faceted navigation provides clues to questions like *What colors, manufacturers, and types of bikes are available?*. Filtering answers questions like *Which exact bikes are red, mountain bikes, in this price range?*. When you click "Red" to indicate that only Red products should be shown, the next query the application sends includes `$filter=Color eq 'Red'`.

The following code snippet from the `JobsSearch.cs` page adds the selected Business Title to the filter if you select a value from the Business Title facet.

```

if (businessTitleFacet != "")
    filter = "business_title eq '" + businessTitleFacet + "'";

```

Tips and best practices

Indexing tips

Improve index efficiency if you don't use a Search box

If your application uses faceted navigation exclusively (that is, no search box), you can mark the field as `searchable=false`, `facetable=true` to produce a more compact index. In addition, indexing occurs only on whole facet values, with no word-break or indexing of the component parts of a multi-word value.

Specify which fields can be used as facets

Recall that the schema of the index determines which fields are available to use as a facet. Assuming a field is facetable, the query specifies which fields to facet by. The field by which you are facetizing provides the values that appear below the label.

The values that appear under each label are retrieved from the index. For example, if the facet field is `Color`, the values available for additional filtering are the values for that field - Red, Black, and so forth.

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`). A values list is allowed for these field types to simplify the separation of facet results into contiguous ranges (either ranges based on numeric values or time periods).

By default you can only have one level of faceted navigation

As noted, there is no direct support for nesting facets in a hierarchy. By default, faceted navigation in Azure Cognitive Search only supports one level of filters. However, workarounds do exist. You can encode a hierarchical facet structure in a `Collection(Edm.String)` with one entry point per hierarchy. Implementing this workaround is beyond the scope of this article.

Querying tips

Validate fields

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.

Filtering tips

Increase search precision with filters

Use filters. If you rely on just search expressions alone, stemming could cause a document to be returned that doesn't have the precise facet value in any of its fields.

Increase search performance with filters

Filters narrow down the set of candidate documents for search and exclude them from ranking. If you have a large set of documents, using a selective facet drill-down often gives you better performance.

Filter only the faceted fields

In faceted drill-down, you typically want to only include documents that have the facet value in a specific (faceted) field, not anywhere across all searchable fields. Adding a filter reinforces the target field by directing the service to search only in the faceted field for a matching value.

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 are not necessarily mutually exclusive. If an item meets the criteria of both filters, it is counted in each one. This duplication is possible when faceting on `Collection(Edm.String)` fields, which are often used to implement document tagging.

```
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.

Tips about result count

Limit the number of items in the facet navigation

For each faceted field in the navigation tree, there is a default limit of 10 values. 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.

- `&facet=city,count:5` specifies that only the first five cities found in the top ranked results are returned as a facet result. Consider a sample query with a search term of "airport" and 32 matches. If the query specifies `&facet=city,count:5`, only the first five unique cities with the most documents in the search results are included in the facet results.

Notice the distinction between facet results and search results. Search results are all the documents that match the query. Facet results are the matches for each facet value. In the example, search results include City names that are not in the facet classification list (5 in our example). Results that are filtered out through faceted navigation become visible when you clear facets, or choose other facets besides City.

NOTE

Discussing `count` when there is more than one type can be confusing. The following table offers a brief summary of how the term is used in Azure Cognitive Search API, sample code, and documentation.

- `@colorFacet.count`

In presentation code, you should see a count parameter on the facet, used to display the number of facet results. In facet results, count indicates the number of documents that match on the facet term or range.

- `&facet=City,count:12`

In a facet query, you can set count to a value. The default is 10, but you can set it higher or lower. Setting `count:12` gets the top 12 matches in facet results by document count.

- `"@odata.count"`

In the query response, this value indicates the number of matching items in the search results. On average, it's larger than the sum of all facet results combined, due to the presence of items that match the search term, but have no facet value matches.

Get counts in facet results

When you add a filter to a faceted query, you might want to retain the facet statement (for example, `facet=Rating&$filter=Rating ge 4`). Technically, `facet=Rating` isn't needed, but keeping it returns the counts of facet values for ratings 4 and higher. For example, if you click "4" and the query includes a filter for greater or equal to "4", counts are returned for each rating that is 4 and higher.

Make sure you get accurate facet counts

Under certain circumstances, you might find that facet counts do not match the result sets (see [Faceted navigation in Azure Cognitive Search \(Microsoft Q&A question page\)](#)).

Facet counts can be inaccurate due to the sharding architecture. Every search index has multiple shards, and each shard reports the top N facets by document count, which is then combined into a single result. If some shards have many matching values, while others have fewer, you may find that some facet values are missing or under-counted in the results.

Although this behavior could change at any time, if you encounter this behavior today, you can work around it by artificially inflating the `count:<number>` to a large number to enforce full reporting from each shard. If the value of `count:` is greater than or equal to the number of unique values in the field, you are guaranteed accurate results. However, when document counts are high, there is a performance penalty, so use this option judiciously.

User interface tips

Add labels for each field in facet navigation

Labels are typically defined in the HTML or form (`index.cshtml` in the sample application). There is no API in Azure Cognitive Search for facet navigation labels or any other metadata.

Filter based on a range

Faceting over ranges of values is a common search application requirement. Ranges are supported for numeric data and DateTime values. You can read more about each approach in [Search Documents \(Azure Cognitive Search API\)](#).

Azure Cognitive Search simplifies range construction by providing two approaches for computing a range. For both approaches, Azure Cognitive Search creates the appropriate ranges given the inputs you've provided. For instance, if you specify range values of 10|20|30, it automatically creates ranges of 0-10, 10-20, 20-30. Your application can optionally remove any intervals that are empty.

Approach 1: Use the interval parameter

To set price facets in \$10 increments, you would specify: `&facet=price,interval:10`

Approach 2: Use a values list

For numeric data, you can use a values list. Consider the facet range for a `listPrice` field, rendered as follows:

Prices:

- 0 - 10 (13)
- 10 - 25 (11)
- 25 - 100 (66)
- 100 - 500 (68)
- 500 - 1000 (50)
- 1000 - 2500 (73)
- 2500 - more (13)

To specify a facet range like the one in the preceding screenshot, use a values list:

```
facet=listPrice,values:10|25|100|500|1000|2500
```

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. Azure Cognitive Search does these things as part of faceted navigation. You do not have to write code for structuring each interval.

Build a filter for a range

To filter documents based on a range you select, you can use the `"ge"` and `"lt"` filter operators in a two-part expression that defines the endpoints of the range. For example, if you choose the range 10-25 for a `listPrice` field, the filter would be `$filter=listPrice ge 10 and listPrice lt 25`. In the sample code, the filter expression uses `priceFrom` and `priceTo` parameters to set the endpoints.

```
private string BuildFilter(string color, string category, double? priceFrom, double? priceTo)
{
    string filter = "&$filter=discontinuedDate eq null";

    if (!string.IsNullOrWhiteSpace(color))
    {
        filter += " and color eq '" + EscapeODataString(color) + "'";
    }

    if (!string.IsNullOrWhiteSpace(category))
    {
        filter += " and categoryName eq '" + EscapeODataString(category) + "'";
    }

    if (priceFrom.HasValue)
    {
        filter += " and listPrice ge " + priceFrom.Value.ToString(CultureInfo.InvariantCulture);
    }

    if (priceTo.HasValue && priceTo > 0)
    {
        filter += " and listPrice le " + priceTo.Value.ToString(CultureInfo.InvariantCulture);
    }
}

return filter;
```

Filter based on distance

It's common to see filters that help you choose a store, restaurant, or destination based on its proximity to your current location. While this type of filter might look like faceted navigation, it's just a filter. We mention it here for those of you who are specifically looking for implementation advice for that particular design problem.

There are two Geospatial functions in Azure Cognitive Search, `geo.distance` and `geo.intersects`.

- The `geo.distance` function returns the distance in kilometers between two points. One point is a field and

other is a constant passed as part of the filter.

- The `geo.intersects` function returns true if a given point is within a given polygon. The point is a field and the polygon is specified as a constant list of coordinates passed as part of the filter.

You can find filter examples in [OData expression syntax \(Azure Cognitive Search\)](#).

Try the demo

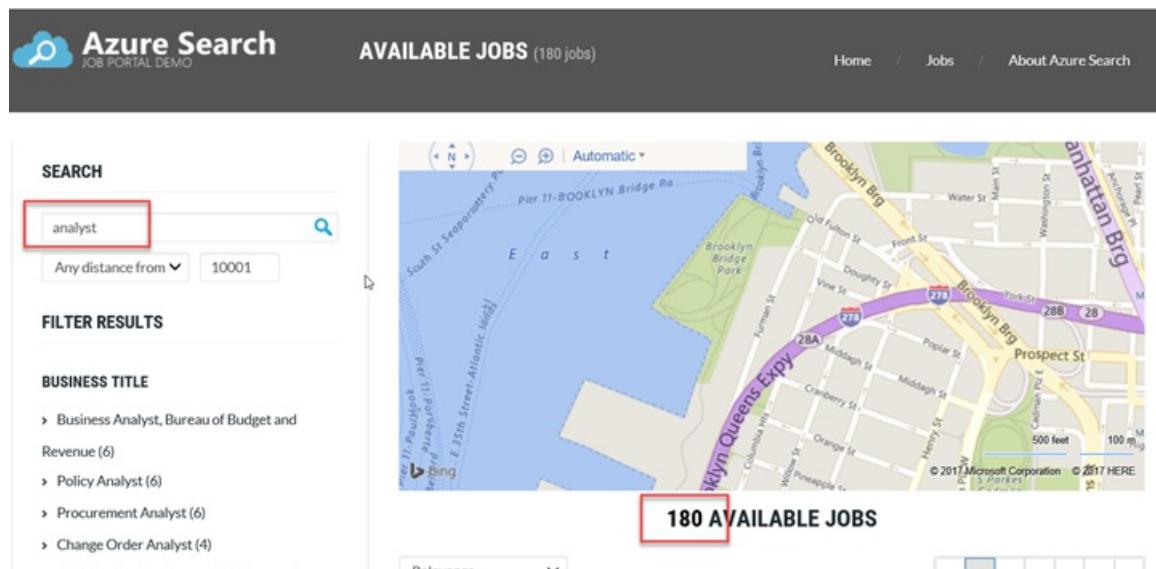
The Azure Cognitive Search Job Portal Demo contains the examples referenced in this article.

- See and test the working demo online at [Azure Cognitive Search Job Portal Demo](#).
- Download the code from the [Azure-Samples repo on GitHub](#).

As you work with search results, watch the URL for changes in query construction. This application happens to append facets to the URI as you select each one.

1. To use the mapping functionality of the demo app, get a Bing Maps key from the [Bing Maps Dev Center](#). Paste it over the existing key in the `index.cshtml` page. The `BingApiKey` setting in the `Web.config` file is not used.
2. Run the application. Take the optional tour, or dismiss the dialog box.
3. Enter a search term, such as "analyst", and click the Search icon. The query executes quickly.

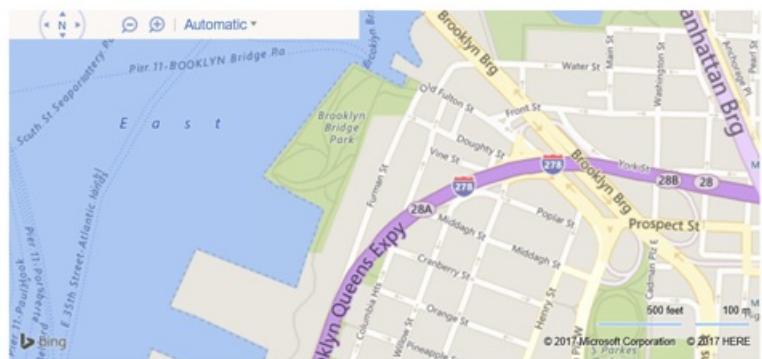
A faceted navigation structure is also returned with the search results. In the search result page, the faceted navigation structure includes counts for each facet result. No facets are selected, so all matching results are returned.



4. Click a Business Title, Location, or Minimum Salary. Facets were null on the initial search, but as they take on values, the search results are trimmed of items that no longer match.

SEARCH

 Any distance from miles



6 AVAILABLE JOBS

FILTER RESULTS

Current Filters:

- Policy Analyst
- \$0 - \$49,999

BUSINESS TITLE

- > Policy Analyst (6)

LOCATION

Brooklyn, NY

5. To clear the faceted query so that you can try different query behaviors, click the after the selected facets to clear the facets.

Learn more

Watch [Azure Cognitive Search Deep Dive](#). At 45:25, there is a demo on how to implement facets.

For more insights on design principles for faceted navigation, we recommend the following links:

- [Design Patterns: Faceted Navigation](#)
- [Front End Concerns When Implementing Faceted Search – Part 1](#)

Troubleshooting OData collection filters in Azure Cognitive Search

10/4/2020 • 11 minutes to read • [Edit Online](#)

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	FOR MORE INFORMATION, SEE
The function 'ismatch' has no parameters bound to the range variable 's'. Only bound field references are supported inside lambda expressions ('any' or 'all'). Please change your filter so that the 'ismatch' function is outside the lambda expression and try again.	Using <code>search.ismatch</code> or <code>search.ismatchscoring</code> 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 <code>Collection(Edm.String)</code> . For 'any', please use expressions of the form 'x eq y' or <code>search.in(...)</code> . For 'all', please use expressions of the form 'x ne y', 'not (x eq y)', or 'not <code>search.in(...)</code> '.	Filtering on a field of type <code>Collection(Edm.String)</code>	Rules for filtering string collections

ERROR MESSAGE	SITUATION	FOR MORE INFORMATION, SEE
<p>Invalid lambda expression. Found an unsupported form of complex Boolean expression. For 'any', please 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', please 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'.</p>	<p>Filtering on fields of type <code>Collection(Edm.DateTimeOffset)</code> , <code>Collection(Edm.Double)</code> , <code>Collection(Edm.Int32)</code> , or <code>Collection(Edm.Int64)</code></p>	Rules for filtering comparable collections
<p>Invalid lambda expression. Found an unsupported use of geo.distance() or geo.intersects() in a lambda expression that iterates over a field of type <code>Collection(Edm.GeographyPoint)</code>. For 'any', make sure you compare geo.distance() using the 'lt' or 'le' operators and make sure that any usage of geo.intersects() is not 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.</p>	<p>Filtering on a field of type <code>Collection(Edm.GeographyPoint)</code></p>	Rules for filtering GeographyPoint collections
<p>Invalid lambda expression. Complex Boolean expressions are not supported in lambda expressions that iterate over fields of type <code>Collection(Edm.GeographyPoint)</code>. For 'any', please join sub-expressions with 'or'; 'and' is not supported. For 'all', please join sub-expressions with 'and'; 'or' is not supported.</p>	<p>Filtering on fields of type <code>Collection(Edm.String)</code> or <code>Collection(Edm.GeographyPoint)</code></p>	Rules for filtering string collections Rules for filtering GeographyPoint collections
<p>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 <code>Collection(Edm.String)</code>. For 'any', please use expressions of the form 'x eq y'. For 'all', please use expressions of the form 'x ne y' or 'not (x eq y)'.</p>	<p>Filtering on a field of type <code>Collection(Edm.String)</code></p>	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 are not 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/all(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)))`
- `locations/all(l: geo.distance(l, geography'POINT(-122 49)') >= 10 and 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 <code>ANY</code>	FEATURES ALLOWED IN LAMBDA EXPRESSIONS WITH <code>ALL</code>
<code>Collection(Edm.ComplexType)</code>	Everything except <code>search.ismatch</code> and <code>search.ismatchscoring</code>	Same
<code>Collection(Edm.String)</code>	Comparisons with <code>eq</code> or <code>search.in</code> Combining sub-expressions with <code>or</code>	Comparisons with <code>ne</code> or <code>not search.in()</code> Combining sub-expressions with <code>and</code>
<code>Collection(Edm.Boolean)</code>	Comparisons with <code>eq</code> or <code>ne</code>	Same
<code>Collection(Edm.GeographyPoint)</code>	Using <code>geo.distance</code> with <code>lt</code> or <code>le</code> <code>geo.intersects</code> Combining sub-expressions with <code>or</code>	Using <code>geo.distance</code> with <code>gt</code> or <code>ge</code> <code>not geo.intersects(...)</code> Combining sub-expressions with <code>and</code>

DATA TYPE	FEATURES ALLOWED IN LAMBDA EXPRESSIONS WITH ANY	FEATURES ALLOWED IN LAMBDA EXPRESSIONS WITH ALL
<code>Collection(Edm.DateTimeOffset)</code> , <code>Collection(Edm.Double)</code> , <code>Collection(Edm.Int32)</code> , <code>Collection(Edm.Int64)</code>	<p>Comparisons using <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>gt</code>, <code>le</code>, or <code>ge</code></p> <p>Combining comparisons with other sub-expressions using <code>or</code></p> <p>Combining comparisons except <code>ne</code> with other sub-expressions using <code>and</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Disjunctive Normal Form (DNF)</p>	<p>Comparisons using <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>gt</code>, <code>le</code>, or <code>ge</code></p> <p>Combining comparisons with other sub-expressions using <code>and</code></p> <p>Combining comparisons except <code>eq</code> with other sub-expressions using <code>or</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Conjunctive Normal Form (CNF)</p>

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

10/4/2020 • 6 minutes to read • [Edit Online](#)

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:

```
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:

```
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:

```
{  
  "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 <code>ROOMS/TYPE</code>	DOCUMENT IDS
deluxe	1, 2
standard	1

How `Rooms/Description` is stored for full-text search:

TERM IN <code>ROOMS/DESCRIPTION</code>	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.

NOTE

If you would like to see support for correlated search added to Azure Cognitive Search, please vote for [this User Voice item](#).

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:

```
{  
  "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:

```
seasons/any(s: s eq 'winter' or s eq 'fall')
```

is equivalent to:

```
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:

```
seasons/all(s: s ne 'winter' and s ne 'fall')
```

is equivalent to:

```
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 \(Azure Cognitive Search REST API\)](#)

How to filter by language in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

A key requirement in a multilingual search application is the ability to search over and retrieve results in the user's own language. In Azure Cognitive Search, one way to meet the language requirements of a multilingual app is to create a series of fields dedicated to storing strings in a specific language, and then constrain full text search to just those fields at query time.

Query parameters on the request are used to both scope the search operation, and then trim the results of any fields that don't provide content compatible with the search experience you want to deliver.

PARAMETERS	PURPOSE
<code>searchFields</code>	Limits full text search to the list of named fields.
<code>\$select</code>	Trims the response to include only the fields you specify. By default, all retrievable fields are returned. The <code>\$select</code> parameter lets you choose which ones to return.

The success of this technique hinges on the integrity of field contents. Azure Cognitive Search does not translate strings or perform language detection. It is up to you to make sure that fields contain the strings you expect.

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 following example is from the [real-estate sample](#) which has several string fields containing content in different languages. Notice the language analyzer assignments for the fields in this index. Fields that contain strings perform better in full text search when paired with an analyzer engineered to handle the linguistic rules of the target language.

Fields
realestate-us-sample

Save Discard

Basic Analyzer Suggester

FIELD NAME	TYPE	SEARCHABLE	ANALYZER
listingId	Edm.String	<input type="checkbox"/>	
beds	Edm.Int32	<input type="checkbox"/>	
baths	Edm.Int32	<input type="checkbox"/>	
description	Edm.String	<input checked="" type="checkbox"/>	English - Microsoft
description_de	Edm.String	<input checked="" type="checkbox"/>	German - Microsoft
description_fr	Edm.String	<input checked="" type="checkbox"/>	French - Microsoft
description_it	Edm.String	<input checked="" type="checkbox"/>	Italian - Microsoft
description_es	Edm.String	<input checked="" type="checkbox"/>	Spanish - Microsoft
description_pl	Edm.String	<input checked="" type="checkbox"/>	Polish - Microsoft
description_nl	Edm.String	<input checked="" type="checkbox"/>	Dutch - Microsoft

NOTE

For code examples showing field definitions with languages analyzers, see [Define an index \(.NET\)](#) and [Define an index \(REST\)](#).

Build and load an index

An intermediate (and perhaps obvious) step is that you have to [build and populate the index](#) before formulating a query. We mention this step here for completeness. One way to determine whether the index is available 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. 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.

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.

```
parameters =
    new SearchParameters()
{
    searchFields = "description_fr"
    Select = new[] { "description_fr" }
};
```

NOTE

Although there is no \$filter argument on the query, this use case is strongly affiliated with filter concepts, so it's presented as a filtering scenario.

See also

- [Filters in Azure Cognitive Search](#)
- [Language analyzers](#)
- [How full text search works in Azure Cognitive Search](#)
- [Search Documents REST API](#)

How to work with search results in Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

This article explains how to get a query response that comes back with a total count of matching documents, paginated results, sorted results, and hit-highlighted terms.

The structure of a response is determined by parameters in the query: [Search Document](#) in the REST API, or [DocumentSearchResult Class](#) in the .NET SDK.

Result composition

While a search document might consist of a large number of fields, typically only a few are needed to represent each document in the result set. On a query request, append `$select=<field list>` to specify which fields show up in the response. A field must be attributed as **Retrievable** in the index to be included in a result.

Fields that work best include those that contrast and differentiate 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 hotels-sample-index built-in sample, it might be fields in the following example:

```
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

If want to include image files in a result, such as a product photo or logo, store them outside of Azure Cognitive Search, but include a field in your index to reference the image URL in the search document. Sample indexes that support images in the results include the `realestate-sample-us` demo, featured in this [quickstart](#), and the [New York City Jobs demo app](#).

Paging results

By default, the search engine returns up to the first 50 matches, as determined by search score if the query is full text search, or in an arbitrary order for exact match queries.

To return a different number of matching documents, add `$top` and `$skip` parameters to the query request. The following list explains the logic.

- Add `$count=true` to get a count of the total number of matching documents within an index.
- Return the first set of 15 matching documents plus a count of total matches:
`GET /indexes/<INDEX-NAME>/docs?search=<QUERY STRING>&$top=15&$skip=0&$count=true`
- Return the second set, skipping the first 15 to get the next 15: `$top=15&$skip=15`. Do the same for the third set of 15: `$top=15&$skip=30`

The results of paginated queries are not 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 is 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:

```
{ "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:

```
{ "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:

```
{ "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

For full text search queries, results are automatically ranked by a search score, calculated based on term frequency and proximity in a document, with higher scores going to documents having more or stronger matches on a search term.

Search scores convey general sense of relevance, reflecting the strength of match as compared to other documents in the same result set. Scores are not 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.

CAUSE	DESCRIPTION
Identical scores	If multiple documents have the same score, any one of them might appear first.

Consistent ordering

Given the flex in results ordering, you might want to explore other options if consistency is an application requirement. The easiest approach is sorting by a field value, such as rating or date. For scenarios where you want to sort by a specific field, such as a rating or date, you can explicitly define an `$orderby` expression, which can be applied to any field that is indexed as `Sortable`.

Another option 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 additional 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. Hit highlighting instructions are provided on the [query request](#).

To enable hit highlighting, add `highlight=[comma-delimited list of string fields]` to specify which fields will use highlighting. Highlighting is useful for longer content fields, such as a description field, where the match is not immediately obvious. Only field definitions that are attributed as `searchable` qualify for hit highlighting.

By default, Azure Cognitive Search returns up to five highlights per field. You can adjust this number by appending to the field a dash followed by an integer. For example, `highlight=Description-10` returns up to 10 highlights on matching content in the Description field.

Formatting is applied to whole term queries. The type of formatting is determined by tags, `highlightPreTag` and `highlightPostTag`, and your code handles the response (for example, applying a bold font or a yellow background).

In the following example, the terms "sandy", "sand", "beaches", "beach" found within the Description field are tagged for highlighting. Queries that trigger query expansion in the engine, such as fuzzy and wildcard search, have limited support for hit highlighting.

```
GET /indexes/hotels-sample-index/docs/search=search=sandy%20beaches&highlight=Description?api-version=2020-06-30
```

```
POST /indexes/hotels-sample-index/docs/search?api-version=2020-06-30
{
    "search": "sandy beaches",
    "highlight": "Description"
}
```

New behavior (starting July 15)

Services created after July 15, 2020 will provide a different highlighting experience. Services created before that date will not change in their highlighting behavior.

With the new behavior:

- Only phrases that match the full phrase query will be returned. The query "super bowl" will return highlights like this:

```
'<em>super bowl</em> is super awesome with a bowl of chips'
```

Note that the term *bowl of chips* does not have any highlighting because it does not match the full phrase.

When you are writing client code that implements hit highlighting, be aware of this change. Note that this will not impact you unless you create a completely new search service.

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.
- [Create your first app in C#](#) is a tutorial that builds a functional client. Sample code demonstrates paginated queries, hit highlighting, and sorting.

Several code samples include a web front-end interface, which you can find here: [New York City Jobs demo app](#), [JavaScript sample code with a live demo site](#), and [CognitiveSearchFrontEnd](#).

Similarity and scoring in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

Scoring refers to the computation of a search score for every item returned in search results for full text search queries. The score is an indicator of an item's relevance in the context of the current search operation. The higher the score, the more relevant the item. In search results, items are rank ordered from high to low, based on the search scores calculated for each item.

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.

The search score is computed based on statistical properties of the data and the query. 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 score values can be repeated throughout a result set. When multiple hits have the same search score, the ordering of the same scored items is not defined, and is 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 = 1.00` 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 a `$filter` expression.

Scoring profiles

You can customize the way different fields are ranked by defining a custom *scoring profile*. Scoring profiles give you greater control over the ranking of items in search results. For example, you might want to boost items 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](#).

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](#)).

```
GET https://[service name].search.windows.net/indexes/[index name]/docs?scoringStatistics=global&api-version=2020-06-30&search=[search term]
Content-Type: application/json
api-key: [admin or query key]
```

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.

```
GET https://[service name].search.windows.net/indexes/[index name]/docs?sessionId=[string]&api-version=2020-06-30&search=[search term]
Content-Type: application/json
api-key: [admin or query key]
```

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.

Similarity ranking algorithms

Azure Cognitive Search supports two different similarity ranking algorithms: A *classic similarity* algorithm and the official implementation of the *Okapi BM25* algorithm (currently in preview). The classical similarity algorithm is the default algorithm, but starting July 15, any new services created after that date use the new BM25 algorithm. It will be the only algorithm available on new services.

For now, you can specify which similarity ranking algorithm you would like to use. For more information, see [Ranking algorithm](#).

The following video segment fast-forwards to an explanation of the ranking algorithms used in Azure Cognitive Search. You can watch the full video for more background.

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:

```
"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](#)

Add scoring profiles to an Azure Cognitive Search index

10/4/2020 • 15 minutes to read • [Edit Online](#)

Scoring computes a search score for each item in a rank ordered result set. Every item in a search result set is assigned a search score, then ranked highest to lowest.

Azure Cognitive Search uses default scoring to compute an initial score, but you can customize the calculation through a *scoring profile*. Scoring profiles give you greater control over the ranking of items in search results. For example, you might want to boost items based on their revenue potential, promote newer items, or perhaps boost items that have been in inventory too long.

The following video segment fast-forwards to how scoring profiles work in Azure Cognitive Search.

Scoring profile definitions

A scoring profile is part of the index definition, composed of weighted fields, functions, and parameters.

To give you an idea of what a scoring profile looks like, the following example shows a simple profile named 'geo'. This one boosts items that have the search term in the **hotelName** field. It also uses the `distance` function to favor items that are within ten 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.

```
"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 the profile on the query string. In the query below, notice the query parameter `scoringProfile=geo` in the request.

```
GET /indexes/hotels/docs?search=inn&scoringProfile=geo&scoringParameter=currentLocation--  
122.123.44.77233&api-version=2020-06-30
```

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 are described in [Search Documents \(Azure Cognitive Search REST API\)](#).

Click [Example](#) to review a more detailed example of a scoring profile.

What is default scoring?

Scoring computes a search score for each item in a rank ordered result set. Every item in a search result set is assigned a search score, then ranked highest to lowest. Items with the higher scores are returned to the application. By default, the top 50 are returned, but you can use the `$top` parameter to return a smaller or larger number of items (up to 1000 in a single response).

The search score is computed based on statistical properties of the data and the query. Azure Cognitive Search finds documents that include the search terms in the query string (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.

Assuming there is no custom sorting, results are then ranked by search score before they are returned to the calling application. If `$top` is not specified, 50 items having the highest search score are returned.

Search score values can be repeated throughout a result set. For example, you might have 10 items with a score of 1.2, 20 items with a score of 1.0, and 20 items with a score of 0.5. When multiple hits have the same search score, the ordering of same scored items is not defined, and is not stable. Run the query again, and you might see items shift position. Given two items with an identical score, there is no guarantee which one appears first.

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 hits that bring benefits to your business can be an important factor in deciding to use scoring profiles.

Relevancy-based ordering 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 drive the 'relevance' option. The definition of relevance is controlled by you, predicated on business objectives and the type of search experience you want to deliver.

Example

As noted earlier, customized scoring is implemented through one or more scoring profiles defined in an index schema.

This 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.

```
{
  "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" ]
    }
  ]
}
```

Workflow

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.

Start with the [Template](#) provided in this topic.

Provide a name. Scoring profiles are optional, but if you add one, the name is required. Be sure to follow the naming conventions for fields (starts with a letter, avoids special characters and reserved words). See [Naming rules \(Azure Cognitive Search\)](#) for the complete list.

The body of the scoring profile is constructed from weighted fields and functions.

Weights	Specify name-value pairs that assign a relative weight to a field. In the Example , the albumTitle, genre, and artistName 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 <code>musicstoreindex</code>), you might need a larger variance in the relative weights. For example, in the <code>musicstoreindex</code> , '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.
----------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Functions

Used when additional calculations are required for specific contexts. Valid values are `freshness`, `magnitude`, `distance`, and `tag`. Each function has parameters that are unique to it.

- `freshness` should be used when you want to boost by how new or old an item is. This function can only be used with `datetime` fields (`edm.DateTimeOffset`). Notice the `boostingDuration` attribute is used only with the `freshness` function.

- `magnitude` should be used when you want to boost 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 double and integer 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` should be used when you want to boost by proximity or geographic location. This function can only be used with `Edm.GeographyPoint` fields.

- `tag` should be used when you want to boost by tags in common between documents and search queries. This function can only be used with `Edm.String` and `Collection(Edm.String)` fields.

Rules for using functions

Function type (`freshness`, `magnitude`, `distance`), `tag` must be lower case.

Functions cannot include null or empty values. Specifically, if you include `fieldname`, you have to set it to something.

Functions can only be applied to filterable fields. See [Create Index \(Azure Cognitive Search REST API\)](#) for more information about filterable fields.

Functions can only be applied to fields that are defined in the `fields` collection of an index.

After the index is defined, build the index by uploading the index schema, followed by documents. See [Create Index \(Azure Cognitive Search REST API\)](#) and [Add, Update or Delete Documents \(Azure Cognitive Search REST API\)](#) for instructions on these operations. Once the index is built, you should have a functional scoring profile that works with your search data.

Template

This section shows the syntax and template for scoring profiles. Refer to [Index attributes reference](#) in the next section for descriptions of the attributes.

```

    ...
  "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) "...",
  ...

```

Index attributes reference

NOTE

A scoring function can only be applied to fields that are filterable.

ATTRIBUTE

DESCRIPTION

ATTRIBUTE	DESCRIPTION
<code>name</code>	Required. This is the name of the scoring profile. It follows the same naming conventions of a field. It must start with a letter, cannot contain dots, colons or @ symbols, and cannot start with the phrase 'azureSearch' (case-sensitive).
<code>text</code>	Contains the weights property.
<code>weights</code>	<p>Optional. Contains name-value pairs that each specify a field name and relative weight. Relative weight must be a positive integer or floating-point number.</p> <p>Weights are used to indicate the importance of one searchable field relative to another.</p>
<code>functions</code>	Optional. A scoring function can only be applied to fields that are filterable.
<code>type</code>	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.
<code>boost</code>	Required for scoring functions. A positive number used as multiplier for raw score. It cannot be equal to 1.
<code>fieldname</code>	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.
<code>interpolation</code>	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.
<code>magnitude</code>	<p>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:</p> <ul style="list-style-type: none"> - 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.

ATTRIBUTE	DESCRIPTION
<code>magnitude boostingRangeStart</code>	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.
<code>magnitude boostingRangeEnd</code>	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.
<code>magnitude constantBoostBeyondRange</code>	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.
<code>freshness</code>	<p>The freshness scoring function is used to alter ranking scores for items based on values in <code>DateTimeOffset</code> fields. For example, an item with a more recent date can be ranked higher than older items.</p> <p>It is 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.</p> <p>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 <code>boostingDuration</code>. To boost a range of times in the future, use a negative <code>boostingDuration</code>.</p> <p>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.</p>
<code>freshness boostingDuration</code>	Sets an expiration period after which boosting will stop for a particular document. See Set boostingDuration in the following section for syntax and examples.
<code>distance</code>	The distance scoring function is used to affect the score of documents based on how close or far they are relative to a reference geographic location. The reference location is given as part of the query in a parameter (using the <code>scoringParameterquery</code> string option) as a lon,lat argument.
<code>distance referencePointParameter</code>	A parameter to be passed in queries to use as reference location. <code>scoringParameter</code> is a query parameter. See Search Documents (Azure Cognitive Search REST API) for descriptions of query parameters.
<code>distance boostingDistance</code>	A number that indicates the distance in kilometers from the reference location where the boosting range ends.

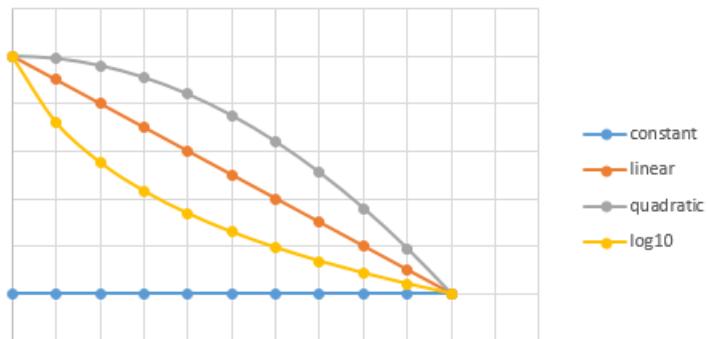
ATTRIBUTE	DESCRIPTION
<code>tag</code>	The tag scoring function is used to affect the score of documents based on tags in documents and search queries. Documents that have tags in common with the search query will be boosted. The tags for the search query is provided as a scoring parameter in each search request (using the <code>scoringParameterquery</code> string option).
<code>tag</code> <code>tagsParameter</code>	A parameter to be passed in queries to specify tags for a particular request. <code>scoringParameter</code> is a query parameter. See Search Documents (Azure Cognitive Search REST API) for descriptions of query parameters.
<code>functionAggregation</code>	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 <code>tf-idf</code> value computed from the document and the search query.
<code>defaultScoringProfile</code>	When executing a search request, if no scoring profile is specified, then default scoring is used (<code>tf-idf</code> only). A default scoring profile name can be set here, causing Azure Cognitive Search to use that profile 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
<code>linear</code>	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.
<code>constant</code>	For items that are within the start and ending range, a constant boost will be applied to the rank results.
<code>quadratic</code>	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 is not allowed in tag scoring functions.
<code>logarithmic</code>	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 is not allowed in tag scoring functions.

Scoring Function Interpolation



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 \(W3.org web site\)](#).

See also

- [REST API Reference](#)
- [Create Index API](#)
- [Azure Cognitive Search .NET SDK](#)

Ranking algorithm in Azure Cognitive Search

10/4/2020 • 4 minutes to read • [Edit Online](#)

IMPORTANT

Starting July 15, 2020, newly created search services will use the BM25 ranking function automatically, which has proven in most cases to provide search rankings that align better with user expectations than the current default ranking. Beyond superior ranking, BM25 also enables configuration options for tuning results based on factors such as document size.

With this change, you will most likely see slight changes in the ordering of your search results. For those who want to test the impact of this change, the BM25 algorithm is available in the api-version 2019-05-06-Preview and in 2020-06-30.

This article describes how you can use the new BM25 ranking algorithm on existing search services for new indexes created and queried using the preview API.

Azure Cognitive Search is in the process of adopting the official Lucene implementation of the Okapi BM25 algorithm, *BM25Similarity*, which will replace the previously used *ClassicSimilarity* implementation. Like the older *ClassicSimilarity* algorithm, BM25Similarity is a TF-IDF-like retrieval function that uses 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.

While conceptually similar to the older Classic Similarity algorithm, BM25 takes its root in probabilistic information retrieval to improve upon it. BM25 also offers advanced customization options, such as allowing the user to decide how the relevance score scales with the term frequency of matched terms.

How to test BM25 today

When you create a new index, you can set a **similarity** property to specify the algorithm. You can use the

`api-version=2019-05-06-Preview`, as shown below, or `api-version=2020-06-30`.

```
PUT https://[search service name].search.windows.net/indexes/[index name]?api-version=2019-05-06-Preview
```

```
{
  "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"
  }
}
```

The **similarity** property is useful during this interim period when both algorithms are available, on existing services only.

PROPERTY	DESCRIPTION
similarity	Optional. Valid values include "#Microsoft.Azure.Search.ClassicSimilarity" or "#Microsoft.Azure.Search.BM25Similarity". Requires <code>api-version=2019-05-06-Preview</code> or later on a search service created prior to July 15, 2020.

For new services created after July 15, 2020, BM25 is used automatically and is the sole similarity algorithm. If you try to set **similarity** to `ClassicSimilarity` on a new service, a 400 error will be returned because that algorithm is not supported on a new service.

For existing services created before July 15, 2020, the Classic similarity remains the default algorithm. If the **similarity** property is omitted or set to null, the index uses the Classic algorithm. If you want to use the new algorithm, you will need to set **similarity** as described above.

BM25 similarity parameters

BM25 similarity adds two user customizable parameters to control the calculated relevance score.

k1

The *k1* parameter controls the scaling function between the term frequency of each matching terms to the final relevance score of a document-query pair.

A value of zero 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. By default, Azure Cognitive Search uses a value of 1.2 for the k1 parameter. 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 which contains the term "Apollo" a few dozen times, without mentions of "Spaceflight", compared to another article which explicitly mentions both "Apollo" and "Spaceflight" a handful of times only.

b

The *b* parameter controls how the length of a document affects the relevance score.

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. The default value used in Azure Cognitive Search for the b parameter is 0.75. 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.

Setting k1 and b parameters

To customize the b or k1 values, simply add them as properties to the similarity object when using BM25:

```
"similarity": {  
    "@odata.type": "#Microsoft.Azure.Search.BM25Similarity",  
    "b" : 0.5,  
    "k1" : 1.3  
}
```

The similarity algorithm can only be set at index creation time. This means the similarity algorithm being used cannot be changed for existing indexes. The "*b*" and "*k1*" parameters can be modified when updating an existing index definition that uses BM25. Changing those values on an existing index will take the index offline for at least a few seconds, causing your indexing and query requests to fail. Because of that, you will need to set the "allowIndexDowntime=true" parameter in the query string of your update request:

```
PUT https://[search service name].search.windows.net/indexes/[index name]?api-version=[api-version]&allowIndexDowntime=true
```

See also

- [REST API Reference](#)
- [Add scoring profiles to your index](#)
- [Create Index API](#)
- [Azure Cognitive Search .NET SDK](#)

Choose a pricing tier for Azure Cognitive Search

10/4/2020 • 15 minutes to read • [Edit Online](#)

When you create an Azure Cognitive Search service, a [resource is created](#) at a pricing tier (or SKU) that's fixed for the lifetime of the service. Tiers include Free, Basic, Standard, and Storage Optimized. Standard and Storage Optimized are available with several configurations and capacities.

Most customers start with the Free tier so they can evaluate the service. Post-evaluation, it's common to create a second service at one of the higher tiers for development and production deployments.

Feature availability by tier

The following table describes tier-related feature constraints.

FEATURE	LIMITATIONS
Indexers	Indexers are not available on S3 HD.
AI enrichment	Runs on the Free tier but not recommended.
Customer-managed encryption keys	Not available on the Free tier.
IP firewall access	Not available on the Free tier.
Integration with Azure Private Link	Not available on the Free tier.

Most features are available on every tier, including Free, but 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.

Tiers (SKUs)

Tiers are differentiated by:

- Quantity of indexes and indexers (maximum limits)
- Size and speed of partitions (physical storage)

The tier you select determines the billable rate. The following screenshot from Azure portal shows the available tiers, minus pricing (which you can find in the portal and on the [pricing page](#). **Free**, **Basic**, and **Standard** are the most common tiers.

Free creates a limited search service for smaller projects, including quickstarts and tutorials. Internally, replicas and partitions are shared among multiple subscribers. You cannot scale a free service or run significant workloads.

Basic and **Standard** are the most commonly used billable tiers, with **Standard** being the default. With dedicated resources under your control, you can deploy larger projects, optimize performance, and set the capacity.

Select Pricing Tier							
Browse available skus and their features							
Sku	Offering	Indexes	Indexers	Storage	Search units	Replicas	Partitions
F	Free	3	3	50 MB	1	1	1
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

Some tiers are optimized for certain types of work. For example, **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 tiers offer larger storage capacity at a lower price per TB than the Standard tiers. The primary tradeoff is higher query latency, which you should validate for your specific application requirements. To learn more about the performance considerations of this tier, see [Performance and optimization considerations](#).

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.

Billable events

A solution built on Azure Cognitive Search can incur costs in the following ways:

- Cost of the service itself, running 24x7, at minimum configuration (one partition and replica)
- Adding capacity (replicas or partitions)
- Bandwidth charges (outbound data transfer)
- Add-on services required for specific capabilities or features:
 - AI enrichment (requires [Cognitive Services](#))
 - knowledge store (requires [Azure Storage](#))
 - incremental enrichment (requires [Azure Storage](#), applies to AI enrichment)
 - customer-managed keys and double encryption (requires [Azure Key Vault](#))
 - private endpoints for a no-internet access model (requires [Azure Private Link](#))

Service costs

Unlike virtual machines or other resources that can be "paused" to avoid charges, an Azure Cognitive Search service is always available on hardware dedicated for your exclusive use. As such, creating a service is a billable event that starts when you create the service, and ends when you delete the service.

The minimum charge is the first search unit (one replica x one partition) at the billable rate. This minimum is fixed for the lifetime of the service because the service can't run on anything less than this configuration. Beyond the minimum, you can add replicas and partitions independently of each other. Incremental increases in capacity through replicas and partitions will increase your bill based on the following formula: ([replicas x partitions x rate](#)), where the rate you're charged depends on the pricing tier you select.

When you're estimating the cost of a search solution, keep in mind that pricing and capacity aren't linear.

(Doubling capacity more than doubles the cost.) For an example of how the formula works, see [How to allocate replicas and partitions](#).

Bandwidth charges

Using [indexers](#) might affect billing, depending on the location of your services. You can eliminate data egress charges entirely if you create the Azure Cognitive Search service in the same region as your data. Here's some information from the [bandwidth pricing page](#):

- Microsoft doesn't charge for any inbound data to any service on Azure, or for any outbound data from Azure Cognitive Search.
- In multiservice solutions, there's no charge for data crossing the wire when all services are in the same region.

Charges do apply for outbound data if services are in different regions. These charges aren't actually part of your Azure Cognitive Search bill. They're mentioned here because if you're using data or AI-enriched indexers to pull data from different regions, you'll see costs reflected in your overall bill.

AI enrichment with Cognitive Services

For [AI enrichment](#), you should plan to [attach a billable Azure Cognitive Services resource](#), in the same region as Azure Cognitive Search, at the S0 pricing tier for pay-as-you-go processing. There's no fixed cost associated with attaching Cognitive Services. You pay only for the processing you need.

OPERATION	BILLING IMPACT
Document cracking, text extraction	Free
Document cracking, image extraction	Billed according to the number of images extracted from your documents. In an indexer configuration , <code>imageAction</code> is the parameter that triggers image extraction. If <code>imageAction</code> is set to "none" (the default), you won't be charged for image extraction. The rate for image extraction is documented on the pricing details page for Azure Cognitive Search.
Built-in cognitive skills	Billed at the same rate as if you had performed the task by using Cognitive Services directly.
Custom skills	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.

The [incremental enrichment \(preview\)](#) feature allows you to provide a cache that enables the indexer to be more efficient at running only the cognitive skills that are necessary if you modify your skillset in the future, saving you time and money.

Billing formula ($R \times P = SU$)

The most important billing concept to understand for Azure Cognitive Search operations is the *search unit* (SU). Because Azure Cognitive Search depends on both replicas and partitions for indexing and queries, it doesn't make sense to bill by just one or the other. Instead, billing is based on a composite of both.

SU is the product of the *replicas* and *partitions* used by a service: ($R \times P = SU$).

Every service starts with one SU (one replica multiplied by one partition) as the minimum. The maximum for any service is 36 SUs. This maximum can be reached in multiple ways: 6 partitions x 6 replicas, or 3 partitions x 12 replicas, for example. It's common to use less than total capacity (for example, a 3-replica, 3-partition service billed as 9 SUs). See the [Partition and replica combinations](#) chart for valid combinations.

The billing rate is hourly per SU. Each tier has a progressively higher rate. Higher tiers come with larger and

speedier partitions, and this contributes to an overall higher hourly rate for that tier. You can view the rates for each tier on the [pricing details](#) page.

Most customers bring just a portion of total capacity online, holding the rest in reserve. For billing, the number of partitions and replicas that you bring online, calculated by the SU formula, determines what you pay on an hourly basis.

How to manage costs

The following suggestions can help you lower costs or manage costs more effectively:

- Create all resources in the same region, or in as few regions as possible, to minimize or eliminate bandwidth charges.
- Consolidate all services into one resource group, such as Azure Cognitive Search, Cognitive Services, and any other Azure services used in your solution. In the Azure portal, find the resource group and use the **Cost Management** commands for insight into actual and projected spending.
- Consider Azure Web App for your front-end application so that requests and responses stay within the data center boundary.
- Scale up for resource-intensive operations like indexing, and then readjust downwards for regular query workloads. Start with the minimum configuration for Azure Cognitive Search (one SU composed of one partition and one replica), and then monitor user activity to identify usage patterns that would indicate a need for more capacity. If there is a predictable pattern, you might be able to synchronize scale with activity (you would need to write code to automate this).

Additionally, visit [Billing and cost management](#) for built-in tools and features related to spending.

Shutting down a search service on a temporary basis is not possible. Dedicated resources are always operational, allocated for your exclusive use for the lifetime of your service. Deleting a service is permanent and also deletes its associated data.

In terms of the service itself, the only way to lower your bill is to reduce replicas and partitions to a level that still provides acceptable performance and [SLA compliance](#), or create a service at a lower tier (S1 hourly rates are lower than S2 or S3 rates). Assuming you provision your service at the lower end of your load projections, if you outgrow the service, you can create a second larger-tiered service, rebuild your indexes on the second service, and then delete the first one.

How to evaluate capacity requirements

In Azure Cognitive Search, capacity is structured as *replicas* and *partitions*.

- Replicas are instances of the search service. Each replica hosts one load-balanced copy of an index. For example, a service with six replicas has six copies of every index loaded in the service.
- Partitions store indexes and automatically split searchable data. Two partitions split your index in half, three partitions split it into thirds, and so on. In terms of capacity, *partition size* is the primary differentiating feature among tiers.

NOTE

All Standard and Storage Optimized tiers support [flexible combinations of replicas and partitions](#) so you can [optimize your system for speed or storage](#) by changing the balance. The Basic tier offers up to three replicas for high availability but has only one partition. Free tiers don't provide dedicated resources: computing resources are shared by multiple subscribers.

Evaluating capacity

Capacity and the costs of running the service go hand in hand. Tiers impose limits on two levels: storage and resources. You should think about both because whichever limit you reach first is the effective limit.

Business requirements typically dictate the number of indexes you'll need. For example, you might need a global index for a large repository of documents. Or you might need multiple indexes based on region, application, or business niche.

To determine the size of an index, you have to [build one](#). Its size will be based on imported data 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.

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 SKU 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.
- [Build an initial index in the portal](#) and note its 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 you know you're going to have 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.

Index number and size are equally important to your analysis. This is because maximum limits are reached through full utilization of storage (partitions) or by maximum limits on resources (indexes, indexers, and so forth), whichever comes first. The portal helps you keep track of both, showing current usage and maximum limits side by side on the Overview page.

NOTE

Storage requirements can be inflated if documents contain extraneous data. 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 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 [Azure Cognitive Search performance and optimization](#).

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 don't provide [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 tier evaluation

- 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.

Next steps

Start with a Free tier and build an initial index by using a subset of your data to understand its characteristics. The data structure in Azure Cognitive Search is an inverted index structure. The size and complexity of an inverted index is determined by content. Remember that highly redundant content tends to result in a smaller index than highly irregular content. So content characteristics rather than the size of the dataset determine index storage requirements.

After you have an initial estimate of your index size, [provision a billable service](#) on one of the tiers discussed in this article: Basic, Standard, or Storage Optimized. Relax any artificial constraints on data sizing and [rebuild your index](#) to include all the data that you want to be searchable.

[Allocate partitions and replicas](#) as needed to get the performance and scale you require.

If performance and capacity are fine, you're done. Otherwise, re-create a search service at a different tier that more closely aligns with your needs.

NOTE

If you have questions, post to [StackOverflow](#) or [contact Azure support](#).

Service limits in Azure Cognitive Search

10/4/2020 • 11 minutes to read • [Edit Online](#)

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 does not 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 within a subscription. Each one can be provisioned at a specific tier. You're limited only by the number of services allowed at each tier. For example, you could create up to 12 services at the Basic tier and another 12 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, contact Azure Support.

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 shared, not dedicated, resources. Scale-up is not supported on shared resources.

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	S1	S2	S3	S3 HD	L1	L2
Maximum elements across all complex collections per document ²	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
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 Having a very large number of elements in complex collections per document currently causes high storage utilization. This is a known issue. In the meantime, a limit of 3000 is a safe upper bound for all service tiers. This limit is only enforced for indexing operations that utilize the earliest generally available (GA) API version that supports complex type fields (`2019-05-06`) onwards. To not break clients who might be using earlier preview API versions (that support complex type fields), we will not be enforcing this limit for indexing operations that use these preview API versions. Note that preview API versions are not meant to be used for production scenarios and we highly recommend customers move to the latest GA API version.

Document limits

As of October 2018, there are no longer any document count limits for any new service created at any billable tier (Basic, S1, S2, S3, S3 HD) in any region. Older services created prior to October 2018 may still be subject to document count limits.

To determine whether your service has document limits, use the [GET Service Statistics REST API](#). Document limits are reflected in the response, with `null` indicating no limits.

NOTE

Although there are no document limits imposed by the service, there is a shard limit of approximately 24 billion documents per index on Basic, S1, S2, and S3 search services. For S3 HD, the shard limit is 2 billion documents per index. Each element of a complex collection counts as a separate document in terms of shard 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 cannot 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	1-3 minutes	24 hours	24 hours	24 hours	24 hours	N/A	24 hours	24 hours

RESOURCE	FREE	BASIC	S1	S2	S3	S3 HD	L1	L2
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. For AI indexing that calls into Cognitive Services, free services are limited to 20 free transactions per day, where a transaction is defined as a document that successfully passes through the enrichment pipeline.

² Basic services created before December 2017 have lower limits (5 instead of 15) on indexers, data sources, and skillsets.

³ S3 HD services do not 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.

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

NOTE

Indexers can access resources securely over private endpoints managed via the [shared private link resource API](#) as described in [this how-to guide](#)

RESOURCE	FREE	BASIC	S1	S2	S3	S3 HD	L1	L2
Private endpoint indexer support	No	Yes	Yes	Yes	Yes	No	Yes	Yes
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, and therefore for lower search service tiers setting them to run in the private environment might have adverse impact on performance and stability of the search service.

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

Queries per second (QPS)

QPS estimates must be developed independently by every customer. Index size and complexity, query size and complexity, and the amount of traffic are primary determinants of QPS. There is no way to offer meaningful estimates when such factors are unknown.

Estimates are more predictable when calculated on services running on dedicated resources (Basic and Standard tiers). You can estimate QPS more closely because you have control over more of the parameters. For guidance on how to approach estimation, see [Azure Cognitive Search performance and optimization](#).

For the Storage Optimized tiers (L1 and L2), you should expect a lower query throughput and higher latency than the Standard tiers.

Data limits (AI enrichment)

An [AI enrichment pipeline](#) that makes calls to a Text Analytics resource for [entity recognition](#), [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

Search query and indexing 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 have static request rate limits.

Static rate request limits for operations related to an index:

- List Indexes (GET /indexes): 5 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

API request limits

- Maximum of 16 MB per request¹
- Maximum 8 KB URL length
- Maximum 1000 documents per batch of index uploads, merges, or deletes
- Maximum 32 fields in \$orderby clause
- Maximum search term size is 32,766 bytes (32 KB minus 2 bytes) of UTF-8 encoded text

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 are not otherwise constrained by theoretical limits (see [Supported data types](#) for more information about field composition and restrictions).

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

Adjust the capacity of an Azure Cognitive Search service

10/4/2020 • 9 minutes to read • [Edit Online](#)

Before [provisioning a search service](#) and locking 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.

Capacity is a function of the [tier you choose](#) (tiers determine hardware characteristics), and the replica and partition combination necessary for projected workloads. You can increase or decrease the number of replicas or partitions individually. Depending on the tier and the size of the adjustment, adding or reducing capacity can take anywhere from 15 minutes to several hours.

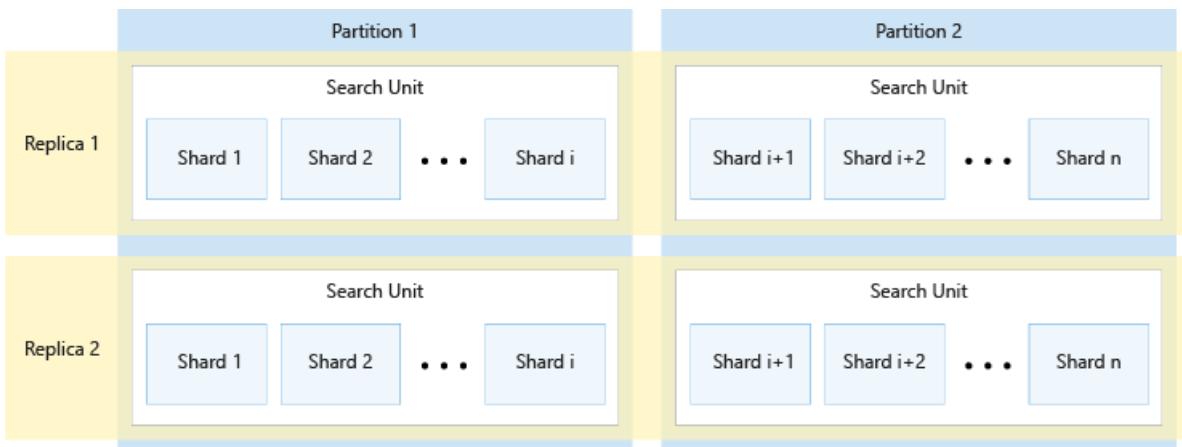
When modifying the allocation of replicas and partitions, we recommend using the Azure portal. The portal enforces limits on allowable combinations that stay below maximum limits of a tier. However, if you require a script-based or code-based provisioning approach, the [Azure PowerShell](#) or the [Management REST API](#) are alternative solutions.

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
<i>Search unit</i>	A single increment of total available capacity (36 units). It is also the billing unit for an Azure Cognitive Search service. A minimum of one unit is required to run the service.
<i>Replica</i>	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.
<i>Partition</i>	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.
<i>Shard</i>	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 non-configurable, 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 unintuitive rankings in search results, it is 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.

When to add nodes

Initially, a service is allocated a minimal level of resources consisting of one partition and one replica.

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 will not otherwise prioritize one task over another. Additionally, a certain amount of redundancy smooths out query performance when services or nodes are updated internally.

As a general rule, search applications tend to need more replicas than partitions, particularly when the service operations are biased toward query workloads. The section on [high availability](#) explains why.

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](#).

How to allocate replicas and partitions

- Sign in to the [Azure portal](#) and select the search service.
- In **Settings**, open the **Scale** page to modify replicas and partitions.

The following screenshot shows a standard service 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.

The screenshot shows the Azure portal interface for configuring a search service. At the top, there are 'Save' and 'Discard' buttons. Below them, under the 'Replicas' section, it says: 'Replicas distribute workloads across the service. We guarantee 99.9% availability for read operations with 2 replicas, and for read and write operations with 3 or more replicas.' A link 'Learn more about the SLA for Azure Search' is provided. A slider for 'Number of replicas' is set to 1, highlighted with a red box. In the 'Partitions' section, it says: 'Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple Azure Search Units (applies to Standard tiers only).' A slider for 'Number of partitions' is also set to 1, highlighted with a red box. Below these sections, the 'Scale Totals' section states: 'Search Unit is the unit of pricing used for Azure Search. It is calculated as the number of partitions x number of replicas. The maximum number of search units is 36.' A callout box highlights the formula '1 Replicas X 1 Partitions' resulting in '1 of 36 available search units'. The entire configuration area is enclosed in a red border.

Replicas

Replicas distribute workloads across the service. We guarantee 99.9% availability for read operations with 2 replicas, and for read and write operations with 3 or more replicas. [Learn more about the SLA for Azure Search](#)

Number of replicas

Partitions

Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple Azure Search Units (applies to Standard tiers only).

Number of partitions

Scale Totals

Search Unit is the unit of pricing used for Azure Search. It is calculated as the number of partitions x number of replicas. The maximum number of search units is 36.

1 of 36 available search units
1 Replicas X 1 Partitions

3. Use the slider to increase or decrease the number of partitions. The formula at the bottom indicates how many search units are being used.

This example doubles capacity, with two replicas and partitions each. Notice the search unit count; it is now four because the billing formula is replicas multiplied by partitions (2×2). 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](#).

 Save

 Discard

Replicas

Replicas distribute workloads across the service. We guarantee 99.9% availability for read operations with 2 replicas, and for read and write operations with 3 or more replicas. [Learn more about the SLA for Azure Search](#)

Number of replicas



2

Partitions

Partitions allow for scaling of document counts as well as faster data ingestion by spanning your index over multiple Azure Search Units (applies to Standard tiers only).

Number of partitions



2

50 GiB Storage

Scale Totals

Search Unit is the unit of pricing used for Azure Search. It is calculated as the number of partitions x number of replicas. The maximum number of search units is 36.

4 of 36 available search units
2 Replicas X 2 Partitions

4. Select Save to confirm the changes.

 Save

 Discard

Update search service

This update may affect the billing. Are you sure you want to update this service?

Yes

No

Changes in capacity take several hours to complete. You cannot cancel once the process has started and there is no real-time monitoring for replica and partition adjustments. However, the following message remains visible while changes are underway.

 Save

 Discard



Updating... This operation may take a while...

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.

Additionally, partitions and replicas are managed exclusively and internally by the service. There is no concept of processor affinity, or assigning a workload to a specific node.

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.

	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
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). This is because 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.

High availability

Because it's easy and relatively fast to scale up, we generally recommend that you start with one partition and one or two replicas, and then scale up as query volumes build. Query workloads run primarily on replicas. If you need more throughput or high availability, you will probably require additional replicas.

General recommendations for high availability are:

- Two replicas for high availability of read-only workloads (queries)
- Three or more replicas for high availability of read/write workloads (queries plus indexing as individual documents are added, updated, or deleted)

Service level agreements (SLA) for Azure Cognitive Search are targeted at query operations and at index updates that consist of adding, updating, or deleting documents.

Basic tier tops out at one partition and three replicas. If you want the flexibility to immediately respond to fluctuations in demand for both indexing and query throughput, consider one of the Standard tiers. If you find your storage requirements are growing much more rapidly than your query throughput, consider one of the Storage Optimized tiers.

Disaster recovery

Currently, there is no built-in mechanism for disaster recovery. Adding partitions or replicas would be the wrong strategy for meeting disaster recovery objectives. The most common approach is to add redundancy at the service level by setting up a second search service in another region. As with availability during an index rebuild, the redirection or failover logic must come from your code.

Estimate replicas

On a production service, you should allocate three replicas for SLA purposes. If you experience slow query performance, you can add replicas so that additional copies of the index are brought online to support bigger query workloads and to load balance the requests over the multiple replicas.

We do not provide 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 is not strictly linear: adding three replicas does not guarantee triple throughput.

For guidance in estimating QPS for your solution, see [Scale for performance](#) and [Monitor queries](#)

Estimate partitions

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.

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.

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.

Next steps

[Choose a pricing tier for Azure Cognitive Search](#)

Scale for performance on Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

This article describes best practices for advanced scenarios with sophisticated requirements for scalability and availability.

Start with baseline numbers

Before undertaking a larger deployment effort, make sure you know what a typical query load looks like. The following guidelines can help you arrive at baseline query numbers.

1. Pick a target latency (or maximum amount of time) that a typical search request should take to complete.
2. Create and test a real workload against your search service with a realistic data set to measure these latency rates.
3. Start with a low number of queries per second (QPS) and then gradually increase the number executed in the test until the query latency drops below the predefined target. This is an important benchmark to help you plan for scale as your application grows in usage.
4. Wherever possible, reuse HTTP connections. If you are using the Azure Cognitive Search .NET SDK, this means you should reuse an instance or [SearchIndexClient](#) instance, and if you are using the REST API, you should reuse a single HttpClient.
5. Vary the substance of query requests so that search occurs over different parts of your index. Variation is important because if you continually execute the same search requests, caching of data will start to make performance look better than it might with a more disparate query set.
6. Vary the structure of query requests so that you get different types of queries. Not every search query performs at the same level. For example, a document lookup or search suggestion is typically faster than a query with a significant number of facets and filters. Test composition should include various queries, in roughly the same ratios as you would expect in production.

While creating these test workloads, there are some characteristics of Azure Cognitive Search to keep in mind:

- It is possible overload your service by pushing too many search queries at one time. When this happens, you will see HTTP 503 response codes. To avoid a 503 during testing, start with various ranges of search requests to see the differences in latency rates as you add more search requests.
- Azure Cognitive Search does not run indexing tasks in the background. If your service handles query and indexing workloads concurrently, take this into account by either introducing indexing jobs into your query tests, or by exploring options for running indexing jobs during off peak hours.

TIP

You can simulate a realistic query load using load testing tools. Try [load testing with Azure DevOps](#) or use one of these [alternatives](#).

Scale for high query volume

A service is overburdened when queries take too long or when the service starts dropping requests. If this happens, you can address the problem in one of two ways:

- **Add replicas**

Each replica is a copy of your data, allowing the service to load balance requests against multiple copies. All load balancing and replication of data 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. Replicas can be adjusted either from the [Azure portal](#) or [PowerShell](#).

- **Create a new service at a higher tier**

Azure Cognitive Search comes in a [number of tiers](#) and each one offers different levels of performance. In some cases, you may have so many queries that the tier you are on cannot provide sufficient turnaround, even when replicas are maxed out. In this case, consider moving to a higher performing tier, such as the Standard S3 tier, designed for scenarios having large numbers of documents and extremely high query workloads.

Scale for slow individual queries

Another reason for high latency rates is a single query taking too long to complete. In this case, adding replicas will not help. Two possible options that might help include the following:

- **Increase Partitions**

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 is that slower queries sometimes perform faster due to parallel computing. We have 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.

There can be a maximum of 12 partitions in Standard search service and 1 partition in the Basic search service. Partitions can be adjusted either from the [Azure portal](#) or [PowerShell](#).

- **Limit High Cardinality Fields**

A high cardinality field consists of 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/filterable would count as high cardinality because most of the values from document to document are unique. Wherever possible, limit the number of high cardinality fields.

- **Increase Search Tier**

Moving up to a higher Azure Cognitive Search tier can be another way to improve performance of slow queries. Each higher tier provides faster CPUs and more memory, both of which have a positive impact on query performance.

Scale for availability

Replicas not only help reduce query latency, but can also allow for high availability. With a single replica, you should expect periodic downtime due to server reboots after software updates or for other maintenance events that will occur. As a result, it is important to consider if your application requires high availability of searches (queries) as well as writes (indexing events). Azure Cognitive Search offers SLA options on all the paid search offerings with the following attributes:

- Two replicas for high availability of read-only workloads (queries)
- Three or more replicas for high availability of read-write workloads (queries and indexing)

For more details on this, please visit the [Azure Cognitive Search Service Level Agreement](#).

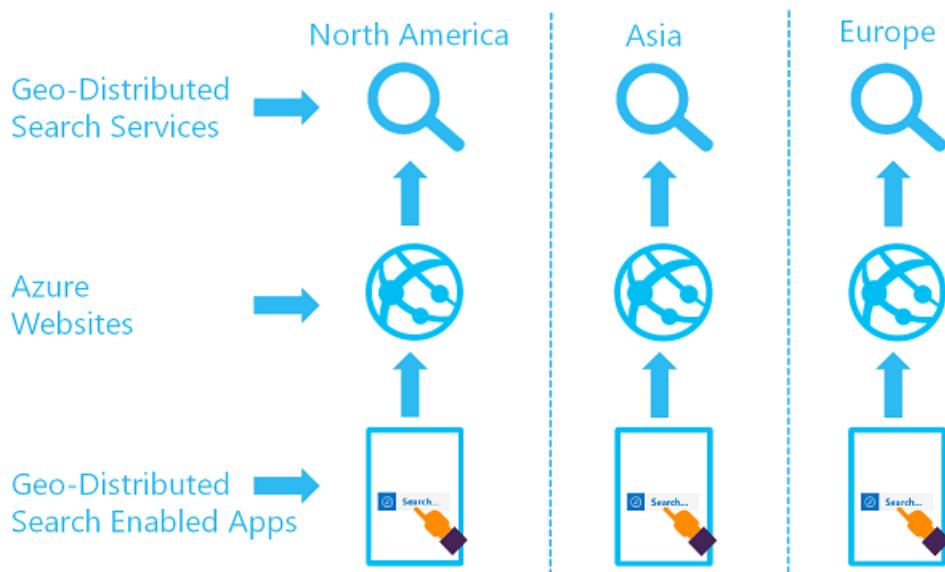
Since replicas are copies of your data, having multiple replicas allows Azure Cognitive Search to do machine reboots and maintenance against one replica, while query execution continues on other replicas. Conversely, if you take replicas away, you'll incur query performance degradation, assuming those replicas were an under-utilized resource.

Scale for geo-distributed workloads and geo-redundancy

For geo-distributed workloads, users who are located far from the host data center will have higher latency rates. One mitigation is to provision multiple search services in regions with closer proximity to these users.

Azure Cognitive Search does not currently provide an automated method of geo-replicating Azure Cognitive Search indexes across regions, but there are some techniques that can be used that can make this process simple to implement and manage. These 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 as seen in this example:



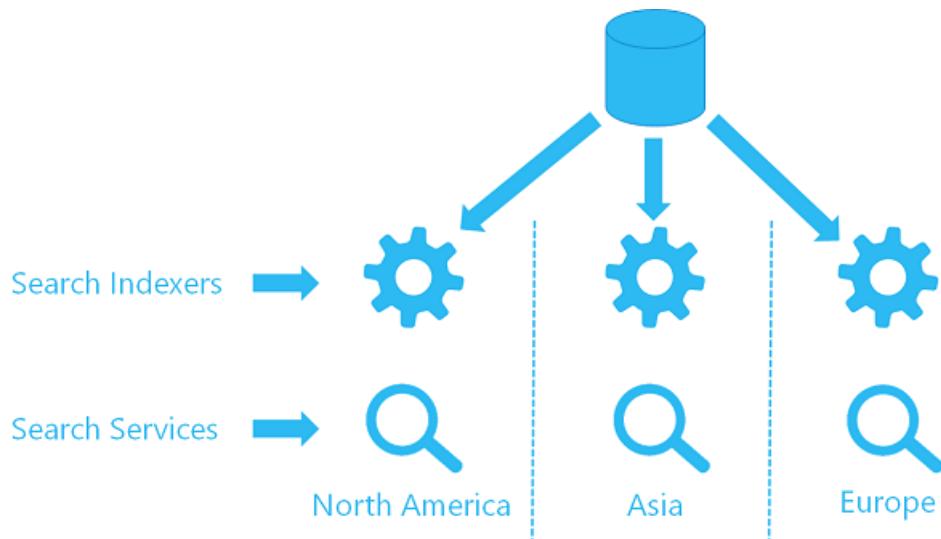
Keep data synchronized across multiple services

There are two options for keeping your distributed search services in sync, which consist of either using the [Azure Cognitive Search Indexer](#) or the Push API (also referred to as the [Azure Cognitive Search REST API](#)).

Use indexers for updating content on multiple services

If you are 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 is not shared, which means data is duplicated), but each indexer references the same data source.

Here is a high-level visual of what that architecture would look like.

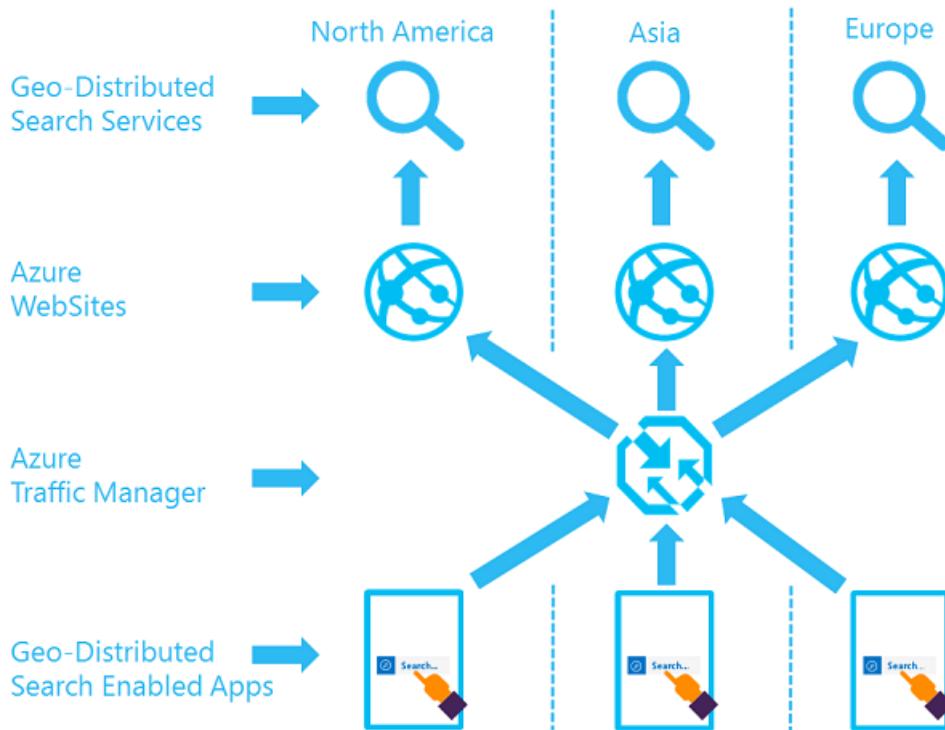


Use REST APIs for pushing content updates on multiple services

If you are using the Azure Cognitive Search REST API to [push content in your Azure Cognitive 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.

Leverage Azure Traffic Manager

[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 ensure that it is available and route users to alternate search services in the event of downtime. In addition, if you are routing search requests through Azure Web Sites, Azure Traffic Manager allows you to load balance cases where the Website is up but not Azure Cognitive Search. Here is an example of what the architecture that leverages Traffic Manager.



Next steps

To learn more about the pricing tiers and services limits for each one, see [Service limits](#). See [Plan for capacity](#) to learn more about partition and replica combinations.

For a discussion about performance and demonstrations of the techniques discussed in this article, watch the following video:

Design patterns for multitenant SaaS applications and Azure Cognitive Search

10/4/2020 • 9 minutes to read • [Edit Online](#)

A multitenant application is one that provides the same services and capabilities to any number of tenants who cannot see or share the data of any other tenant. This document 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 is 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 is 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

	BASIC	STANDARD1	STANDARD2	STANDARD3	STANDARD3 HD
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 is an option for the High Density (HD) mode designed specifically for multitenant scenarios. In many cases, it is 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 is 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.

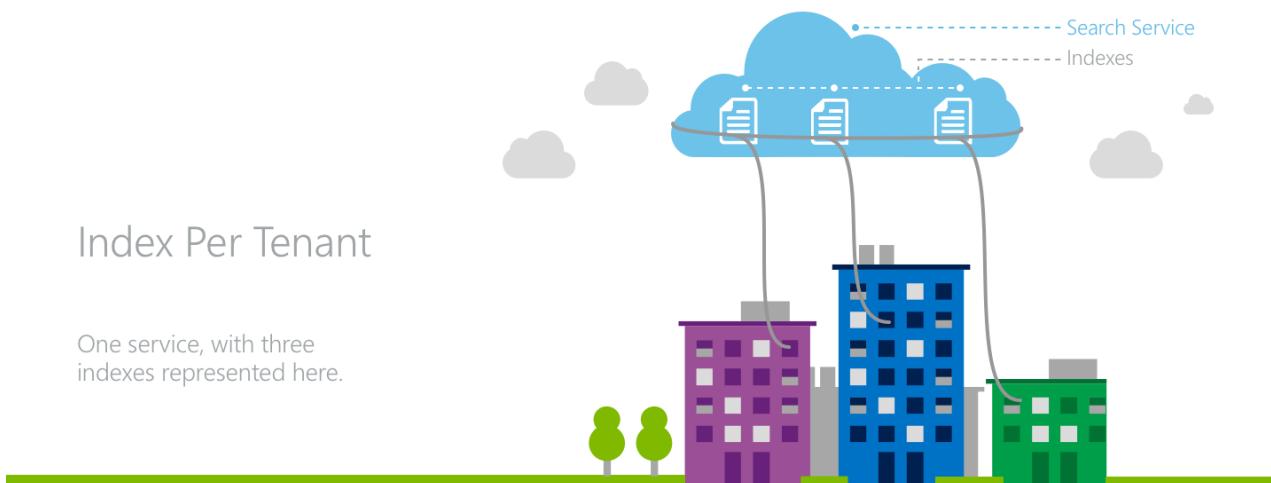
Azure Cognitive Search offers a few boundaries that can be used to isolate tenants' data and workload.

Modeling multitenancy with Azure Cognitive Search

In the case of a multitenant scenario, the application developer consumes one or more search services and divide their tenants among services, indexes, or both. Azure Cognitive Search has a few common patterns when modeling a multitenant scenario:

1. *Index per tenant*: Each tenant has its own index within a search service that is shared with other tenants.
2. *Service per tenant*: Each tenant has its own dedicated Azure Cognitive Search service, offering highest level of data and workload separation.
3. *Mix of both*: Larger, more-active tenants are assigned dedicated services while smaller tenants are assigned individual indexes within shared services.

1. 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 is 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 does not allow for an index to be moved.

2. 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 as well as separate 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 are not shared across various tenants' workloads.

A service per tenant model also offers the benefit of a predictable, fixed cost model. There is no up-front investment in an entire search service until there is 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 is 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 does not currently support upgrading the pricing tier of a search service, so all data would have to be manually copied to a new service.

3. Mixing both models

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 are not sufficiently small scopes, it is 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.

Any questions about Azure Cognitive Search and multitenant scenarios can be directed to azuresearch_contact@microsoft.com.

API versions in Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

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 Azure Cognitive Search team publishes new versions 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.

The same rule applies for SDK updates. The Azure Cognitive Search SDK follows the [semantic versioning](#) rules, which means that its version has three parts: major, minor, and build number (for example, 1.1.0). A new major version of the SDK is released only for changes that break backward compatibility. Non-breaking feature updates will increment the minor version, and bug fixes will only increase the build version.

IMPORTANT

The Azure SDKs for .NET, Java, Python and JavaScript are rolling out new client libraries for Azure Cognitive Search. Currently, none of the Azure SDK libraries fully support the most recent Search REST APIs (2020-06-30) or Management REST APIs (2020-03-13) but this will change over time. You can periodically check this page or the [What's New](#) for announcements on functional enhancements.

Unsupported versions

Upgrade existing search solutions to the latest version of the REST API by October 15, 2020. At that time, the following versions of the Azure Cognitive Search REST API will be retired and no longer supported:

- **2015-02-28**
- **2015-02-28-Preview**
- **2014-07-31-Preview**
- **2014-10-20-Preview**

In addition, versions of the Azure Cognitive Search .NET SDK older than **3.0.0-rc** will also be retired since they target one of these REST API versions.

After this date, applications that use any of the deprecated REST API or SDK versions will no longer work and must be upgraded. As with any change of this type, we are giving 12 months' notice, so you have adequate time to adjust.

To continue using Azure Cognitive Search, please migrate existing code that targets the [REST API](#) to [REST API version 2020-06-30](#) or to a newer SDK by October 15, 2020. If you have any questions about updating to the latest version, please send mail to azuresearch_contact@microsoft.com by May 15, 2020 to ensure you have enough time to update your code.

REST APIs

An Azure Cognitive Search service instance supports several REST API versions, including the latest one. You can continue to use a version when it is no longer the latest one, but we recommend that you [migrate your code](#) to use the newest version. When using the REST API, you must specify the API version in every request via the `api-version`

parameter. When using the .NET SDK, the version of the SDK you're using determines the corresponding version of the REST API. If you are using an older SDK, you can continue to run that code with no changes even if the service is upgraded to support a newer API version.

The following table provides the version history of current and previously released versions of the Search Service REST API. Documentation is published only for the most recent stable and preview versions.

Search Service APIs

Create and manage content on a search service.

VERSION	STATUS	DESCRIPTION
Search 2020-06-30	Stable	Newest stable release of the Search REST APIs, with advancements in relevance scoring and generally availability for knowledge store.
Search 2020-06-30-Preview	Preview	Preview version associated with stable version. Includes multiple preview features .
Search 2019-05-06	Stable	Adds complex types .
Search 2019-05-06-Preview	Preview	Preview version associated with stable version.
Search 2017-11-11	Stable	Adds skillsets and AI enrichment .
Search 2017-11-11-Preview	Preview	Preview version associated with stable version.
Search 2016-09-01	Stable	Adds indexers .
Search 2016-09-01-Preview	Preview	Preview version associated with stable version.
Search 2015-02-28	Unsupported after 10-10-2020	First generally available release.
Search 2015-02-28-Preview	Unsupported after 10-10-2020	Preview version associated with stable version.
Search 2014-10-20-Preview	Unsupported after 10-10-2020	Second public preview.
Search 2014-07-31-Preview	Unsupported after 10-10-2020	First public preview.

Management REST APIs

Create and configure a search service, and manage API keys.

VERSION	STATUS	DESCRIPTION
Management 2020-08-01	Stable	Newest stable release of the Management REST APIs. Adds generally available shared private link resource support for all outbound-accessed resources except those noted in the preview version

VERSION	STATUS	DESCRIPTION
Management 2020-08-01-Preview	Preview	Currently in preview: shared private link resource support for Azure Functions and Azure Database for MySQL.
Management 2020-03-13	Stable	Adds private endpoint through private link, and network IP rules for new services. For more information, see this swagger specification .
Management 2019-10-01-Preview	Preview	There were no preview features introduced in this list. This preview is functionally equivalent to 2020-03-13. For more information, see this swagger specification .
Management 2015-08-19	Stable	The first generally available version of the Management REST APIs. Provides service provisioning, scale up, and api-key management. For more information, see this swagger specification .
Management 2015-08-19-Preview	Preview	The first preview version of the Management REST APIs. For more information, see this swagger specification .

Azure SDK for .NET

The following table provides links to more recent SDK versions.

SDK VERSION	STATUS	DESCRIPTION
Azure.Search.Documents 11.0	Stable	New client library from Azure .NET SDK, released July 2020. Targets the Search REST api-version=2020-06-30 REST API but does not yet support, geo-filters, or FieldBuilder .
Microsoft.Azure.Search 10.0	Stable	Released May 2019. Targets the Search REST api-version=2019-05-06.
Microsoft.Azure.Management.Search 4.0.0	Stable	Targets the Management REST api-version=2020-08-01.
Microsoft.Azure.Management.Search 3.0.0	Stable	Targets the Management REST api-version=2015-08-19.

Azure SDK for Java

SDK VERSION	STATUS	DESCRIPTION
Java azure-search-documents 11	Stable	New client library from Azure .NET SDK, released July 2020. Targets the Search REST api-version=2019-05-06.
Java Management Client 1.35.0	Stable	Targets the Management REST api-version=2015-08-19.

Azure SDK for JavaScript

SDK VERSION	STATUS	DESCRIPTION
JavaScript azure-search 11.0	Stable	New client library from Azure .NET SDK, released July 2020. Targets the Search REST api-version=2016-09-01.
JavaScript azure-arm-search	Stable	Targets the Management REST api-version=2015-08-19.

Azure SDK for Python

SDK VERSION	STATUS	DESCRIPTION
Python azure-search-documents 11.0	Stable	New client library from Azure .NET SDK, released July 2020. Targets the Search REST api-version=2019-05-06.
Python azure-mgmt-search 1.0	Stable	Targets the Management REST api-version=2015-08-19.

Preview features in Azure Cognitive Search

10/4/2020 • 4 minutes to read • [Edit Online](#)

This article is a comprehensive list of all features that are in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#).

Preview features that transition to general availability are removed from this list. If a feature isn't listed below, you can assume it is generally available. For announcements regarding general availability, see [Service Updates](#) or [What's New](#).

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Azure Machine Learning (AML) skill	AI enrichment	A new skill type to integrate an inferencing endpoint from Azure Machine Learning. Get started with this tutorial .	Use Search REST API 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.
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 (REST) with api-version=2020-06-30-Preview or 2019-05-06-Preview.
Debug Sessions	Portal, AI enrichment (skillset)	An in-session skillset editor used to investigate and resolve issues with a skillset. Fixes applied during a debug session can be saved to a skillset in the service.	Portal only, using mid-page links on the Overview page to open a debug session.
Native blob soft delete	Indexers, Azure blobs	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.	Add this configuration setting using Create Indexer (REST) with api-version=2020-06-30-Preview or api-version=2019-05-06-Preview.

FEATURE	CATEGORY	DESCRIPTION	AVAILABILITY
Custom Entity Lookup skill	AI enrichment (skillset)	A cognitive skill that looks for text from a custom, user-defined list of words and phrases. Using this list, it labels all documents with any matching entities. The skill also supports a degree of fuzzy matching that can be applied to find matches that are similar but not quite exact.	Reference this preview skill using the Skillset editor in the portal or Create Skillset (REST) with api-version=2020-06-30-Preview or api-version=2019-05-06-Preview.
PII Detection skill	AI enrichment (skillset)	A cognitive skill used during indexing that extracts personal information from an input text and gives you the option to mask it from that text in various ways.	Reference this preview skill using the Skillset editor in the portal or Create Skillset (REST) with api-version=2020-06-30-Preview or api-version=2019-05-06-Preview.
Incremental enrichment	Indexer configuration	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, does not change the content. Caching applies only to enriched documents produced by a skillset.	Add this configuration setting using Create Indexer (REST) with api-version=2020-06-30-Preview or api-version=2019-05-06-Preview.
Cosmos DB indexer: MongoDB API, Gremlin API, Cassandra API	Indexer data source	For Cosmos DB, SQL API is generally available, but MongoDB, Gremlin, and Cassandra APIs are in preview.	For Gremlin and Cassandra only, sign up first so that support can be enabled for your subscription on the backend. MongoDB data sources can be configured in the portal. Otherwise, data source configuration for all three APIs is supported using Create Data Source (REST) with api-version=2020-06-30-Preview or api-version=2019-05-06-Preview.
Azure Data Lake Storage Gen2 indexer	Indexer data source	Index content and metadata from Data Lake Storage Gen2.	Sign up is required so that support can be enabled for your subscription on the backend. Access this data source using Create Data Source (REST) with api-version=2020-06-30-Preview or api-version=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 (REST) calls with api-version=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 do not 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, [2020-06-30-Preview](#) is the current preview version.
- For management operations, [2019-10-01-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. However, only the newest preview version is refreshed with improvements.

The following example syntax illustrates a call to the preview API version.

```
POST https://[service name].search.windows.net/indexes/hotels-idx/docs/search?api-version=2020-06-30-Preview
Content-Type: application/json
api-key: [admin key]
```

Azure Cognitive Search service is available in multiple versions. 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 use Microsoft.Azure.Search (v10) in a .NET Application

10/4/2020 • 24 minutes to read • [Edit Online](#)

This article explains how to create and manage search objects using C# and the [Azure Cognitive Search \(v10\) .NET SDK](#). Version 10 is the last version of the Microsoft.Azure.Search package. Moving forward, new features will be rolled out in [Azure.Search.Documents](#) from the Azure SDK team.

If you have existing or inflight development projects, continue to use version 10. For new projects, or to use new features, you should transition an existing search solution to the new library.

What's in version 10

The SDK consists of a few client libraries that enable you to manage your indexes, data sources, indexers, and synonym maps, as well as upload and manage documents, and execute queries, all without having to deal with the details of HTTP and JSON. These client libraries are all distributed as NuGet packages.

The main NuGet package is `Microsoft.Azure.Search`, which is a meta-package that includes all the other packages as dependencies. Use this package if you're just getting started or if you know your application will need all the features of Azure Cognitive Search.

The other NuGet packages in the SDK are:

- `Microsoft.Azure.Search.Data`: Use this package if you're developing a .NET application using Azure Cognitive Search, and you only need to query or update documents in your indexes. If you also need to create or update indexes, synonym maps, or other service-level resources, use the `Microsoft.Azure.Search` package instead.
- `Microsoft.Azure.Search.Service`: Use this package if you're developing automation in .NET to manage Azure Cognitive Search indexes, synonym maps, indexers, data sources, or other service-level resources. If you only need to query or update documents in your indexes, use the `Microsoft.Azure.Search.Data` package instead. If you need all the functionality of Azure Cognitive Search, use the `Microsoft.Azure.Search` package instead.
- `Microsoft.Azure.Search.Common`: Common types needed by the Azure Cognitive Search .NET libraries. You do not need to use this package directly in your application. It is only meant to be used as a dependency.

The various client libraries define classes like `Index`, `Field`, and `Document`, as well as operations like `Indexes.Create` and `Documents.Search` on the `SearchServiceClient` and `SearchIndexClient` classes. These classes are organized into the following namespaces:

- [Microsoft.Azure.Search](#)
- [Microsoft.Azure.Search.Models](#)

If you would like to provide feedback for a future update of the SDK, see our [feedback page](#) or create an issue on [GitHub](#) and mention "Azure Cognitive Search" in the issue title.

The .NET SDK targets version `2019-05-06` of the [Azure Cognitive Search REST API](#). This version includes support for [complex types](#), [AI enrichment](#), [autocomplete](#), and [JsonLines parsing mode](#) when indexing Azure Blobs.

This SDK does not support [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, you can use the [Azure Cognitive Search .NET Management SDK](#).

Upgrading to the latest version of the SDK

If you're already using an older version of the Azure Cognitive Search .NET SDK and you'd like to upgrade to the latest generally available version, [this article](#) explains how.

Requirements for the SDK

1. Visual Studio 2017 or later.
2. Your own Azure Cognitive Search service. In order to use the SDK, you will need the name of your service and one or more API keys. [Create a service in the portal](#) will help you through these steps.
3. Download the Azure Cognitive Search .NET SDK [NuGet package](#) by using "Manage NuGet Packages" in Visual Studio. Just search for the package name `Microsoft.Azure.Search` on NuGet.org (or one of the other package names above if you only need a subset of the functionality).

The Azure Cognitive Search .NET SDK supports applications targeting the .NET Framework 4.5.2 and higher, as well as .NET Core 2.0 and higher.

Core scenarios

There are several things you'll need to do in your search application. In this tutorial, we'll cover these core scenarios:

- Creating an index
- Populating the index with documents
- Searching for documents using full-text search and filters

The following sample code illustrates each of these scenarios. Feel free to use the code snippets in your own application.

Overview

The sample application we'll be exploring creates a new index named "hotels", populates it with a few documents, then executes some search queries. Here is the main program, showing the overall flow:

```

// 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();

    SearchServiceClient serviceClient = CreateSearchServiceClient(configuration);

    string indexName = configuration["SearchIndexName"];

    Console.WriteLine("{0}", "Deleting index...\\n");
    DeleteIndexIfExists(indexName, serviceClient);

    Console.WriteLine("{0}", "Creating index...\\n");
    CreateIndex(indexName, serviceClient);

    ISearchIndexClient indexClient = serviceClient.Indexes.GetClient(indexName);

    Console.WriteLine("{0}", "Uploading documents...\\n");
    UploadDocuments(indexClient);

    ISearchIndexClient indexClientForQueries = CreateSearchIndexClient(configuration);

    RunQueries(indexClientForQueries);

    Console.WriteLine("{0}", "Complete. Press any key to end application...\\n");
    Console.ReadKey();
}

```

NOTE

You can find the full source code of the sample application used in this walk through on [GitHub](#).

We'll walk through this step by step. First we need to create a new `SearchServiceClient`. This object allows you to manage indexes. In order to construct one, you need to provide your Azure Cognitive Search service name as well as an admin API key. You can enter this information in the `appsettings.json` file of the [sample application](#).

```

private static SearchServiceClient CreateSearchServiceClient(IConfigurationRoot configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string adminApiKey = configuration["SearchServiceAdminApiKey"];

    SearchServiceClient serviceClient = new SearchServiceClient(searchServiceName, new
    SearchCredentials(adminApiKey));
    return serviceClient;
}

```

NOTE

If you provide an incorrect key (for example, a query key where an admin key was required), the `SearchServiceClient` will throw a `CloudException` with the error message "Forbidden" the first time you call an operation method on it, such as `Indexes.Create`. If this happens to you, double-check our API key.

The next few lines call methods to create an index named "hotels", deleting it first if it already exists. We will walk through these methods a little later.

```
Console.WriteLine("{0}", "Deleting index...\n");
DeleteIndexIfExists(indexName, serviceClient);

Console.WriteLine("{0}", "Creating index...\n");
CreateIndex(indexName, serviceClient);
```

Next, the index needs to be populated. To do populate the index, we will need a `SearchIndexClient`. There are two ways to obtain one: by constructing it, or by calling `Indexes.GetClient` on the `SearchServiceClient`. We use the latter for convenience.

```
ISearchIndexClient indexClient = serviceClient.Indexes.GetClient(indexName);
```

NOTE

In a typical search application, index management and population may be handled by a separate component from search queries. `Indexes.GetClient` is convenient for populating an index because it saves you the trouble of providing additional `SearchCredentials`. It does this by passing the admin key that you used to create the `SearchServiceClient` to the new `SearchIndexClient`. However, in the part of your application that executes queries, it is better to create the `SearchIndexClient` directly so that you can pass in a query key, which only allows you to read data, instead of an admin key. This is consistent with the principle of least privilege and will help to make your application more secure. You can find out more about admin keys and query keys [here](#).

Now that we have a `SearchIndexClient`, we can populate the index. Index population is done by another method that we will walk through later.

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

Finally, we execute a few search queries and display the results. This time we use a different `SearchIndexClient`:

```
ISearchIndexClient indexClientForQueries = CreateSearchIndexClient(indexName, configuration);

RunQueries(indexClientForQueries);
```

We will take a closer look at the `RunQueries` method later. Here is the code to create the new `SearchIndexClient`:

```
private static SearchIndexClient CreateSearchIndexClient(string indexName, IConfigurationRoot
configuration)
{
    string searchServiceName = configuration["SearchServiceName"];
    string queryApiKey = configuration["SearchServiceQueryApiKey"];

    SearchIndexClient indexClient = new SearchIndexClient(searchServiceName, indexName, new
SearchCredentials(queryApiKey));
    return indexClient;
}
```

This time we use a query key since we do not need write access to the index. You can enter this information in the `appsettings.json` file of the [sample application](#).

If you run this application with a valid service name and API keys, the output should look like this example: (Some console output has been replaced with "..." for illustration purposes.)

```
Deleting index...
```

```
Creating index...
```

```
Uploading documents...
```

```
Waiting for documents to be indexed...
```

```
Search the entire index for the term 'motel' and return only the HotelName field:
```

```
Name: Secret Point Motel
```

```
Name: Twin Dome Motel
```

```
Apply a filter to the index to find hotels with a room cheaper than $100 per night, and return the hotelId and description:
```

```
HotelId: 1
```

```
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 Times 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.
```

```
HotelId: 2
```

```
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.
```

```
Search the entire index, order by a specific field (lastRenovationDate) in descending order, take the top two results, and show only hotelName and lastRenovationDate:
```

```
Name: Triple Landscape Hotel
```

```
Last renovated on: 9/20/2015 12:00:00 AM +00:00
```

```
Name: Twin Dome Motel
```

```
Last renovated on: 2/18/1979 12:00:00 AM +00:00
```

```
Search the hotel names for the term 'hotel':
```

```
HotelId: 3
```

```
Name: Triple Landscape Hotel
```

```
...
```

```
Complete. Press any key to end application...
```

The full source code of the application is provided at the end of this article.

Next, we will take a closer look at each of the methods called by `Main`.

Creating an index

After creating a `SearchServiceClient`, `Main` deletes the "hotels" index if it already exists. That deletion is done by the following method:

```
private static void DeleteIndexIfExists(string indexPathName, SearchServiceClient serviceClient)
{
    if (serviceClient.Indexes.Exists(indexPathName))
    {
        serviceClient.Indexes.Delete(indexPathName);
    }
}
```

This method uses the given `SearchServiceClient` to check if the index exists, and if so, delete it.

NOTE

The example code in this article uses the synchronous methods of the Azure Cognitive Search .NET SDK for simplicity. We recommend that you use the asynchronous methods in your own applications to keep them scalable and responsive. For example, in the method above you could use `ExistsAsync` and `DeleteAsync` instead of `Exists` and `Delete`.

Next, `Main` creates a new "hotels" index by calling this method:

```
private static void CreateIndex(string indexName, SearchServiceClient serviceClient)
{
    var definition = new Index()
    {
        Name = indexName,
        Fields = FieldBuilder.BuildForType<Hotel>()
    };

    serviceClient.Indexes.Create(definition);
}
```

This method creates a new `Index` object with a list of `Field` objects that defines the schema of the new index. Each field has a name, data type, and several attributes that define its search behavior. The `FieldBuilder` class uses reflection to create a list of `Field` 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.

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.

In addition to fields, you can also add scoring profiles, suggesters, or CORS options to the `Index` object (these parameters are omitted from the sample for brevity). You can find more information about the `Index` object and its constituent parts in the [SDK reference](#), as well as in the [Azure Cognitive Search REST API reference](#).

Populating the index

The next step in `Main` populates the newly-created 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.)

```
private static void UploadDocuments(ISearchIndexClient indexClient)
{
    var hotels = new Hotel[]
    {
        new Hotel()
        {
            HotelId = "1",
            HotelName = "Secret Point Motel",
            ...
            Address = new Address()
            {
                StreetAddress = "677 5th Ave",
                ...
            },
            Rooms = new Room[]
            {
                new Room()
            }
        }
    };
    indexClient.Documents.Index(hotels);
}
```

```

        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)",
        ...
    }
}
},
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)",
            ...
        }
    }
},
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)",
        ...
    }
}
};

var batch = IndexBatch.Upload(hotels);

try
{
    indexClient.Documents.Index(batch);
}
catch (IndexBatchException e)
{
    // 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}",
        String.Join(", ", e.IndexingResults.Where(r => !r.Succeeded).Select(r => r.Key)));
}

Console.WriteLine("Waiting for documents to be indexed...\n");
Thread.Sleep(2000);
}

```

This method has four parts. The first creates an array of 3 `Hotel` objects each with 3 `Room` objects that will serve as our input data to upload to the index. This data is hard-coded for simplicity. In your own application, your data will likely come from an external data source such as a SQL database.

The second part creates an `IndexBatch` 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 `IndexBatch.Upload`. The batch is then uploaded to the Azure Cognitive Search index by the `Documents.Index` 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 `IndexBatch.Merge`, `IndexBatch.MergeOrUpload`, or `IndexBatch.Delete` instead. You can also mix different operations in a single batch by calling `IndexBatch.New`, which takes a collection of `IndexAction` objects, each of which tells Azure Cognitive Search to perform a particular operation on a document. You can create each `IndexAction` with its own operation by calling the corresponding method such as `IndexAction.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 Azure Cognitive Search service fails to index some of the documents in the batch, an `IndexBatchException` is thrown by `Documents.Index`. This exception can happen if you are 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.

NOTE

You can use the `FindFailedActionsToRetry` method to construct a new batch containing only the actions that failed in a previous call to `Index`. There is a discussion of how to properly use it [on StackOverflow](#).

Finally, the `UploadDocuments` method delays for two seconds. Indexing happens asynchronously in your Azure

Cognitive 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.

How the .NET SDK handles documents

You may be wondering how the Azure Cognitive Search .NET SDK is able to upload instances of a user-defined class like `Hotel` to the index. To help answer that question, let's look at the `Hotel` class:

```
using System;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Microsoft.Spatial;
using Newtonsoft.Json;

public partial class Hotel
{
    [System.ComponentModel.DataAnnotations.Key]
    [IsFilterable]
    public string HotelId { get; set; }

    [IsSearchable, IsSortable]
    public string HotelName { get; set; }

    [IsSearchable]
    [Analyzer(AnalyzerName.AsString.EnLucene)]
    public string Description { get; set; }

    [IsSearchable]
    [Analyzer(AnalyzerName.AsString.FrLucene)]
    [JsonProperty("Description_fr")]
    public string DescriptionFr { get; set; }

    [IsSearchable, IsFilterable, IsSortable, IsFacetable]
    public string Category { get; set; }

    [IsSearchable, IsFilterable, IsFacetable]
    public string[] Tags { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public bool? SmokingIncluded { get; set; }

    // SmokingAllowed reflects whether any room in the hotel allows smoking.
    // The JsonIgnore attribute indicates that a field should not be created
    // in the index for this property and it will only be used by code in the client.
    [JsonIgnore]
    public bool? SmokingAllowed => (Rooms != null) ? Array.Exists(Rooms, element => element.SmokingAllowed == true) : (bool?)null;

    [IsFilterable, IsSortable, IsFacetable]
    public DateTimeOffset? LastRenovationDate { get; set; }

    [IsFilterable, IsSortable, IsFacetable]
    public double? Rating { get; set; }

    public Address Address { get; set; }

    [IsFilterable, IsSortable]
    public GeographyPoint Location { get; set; }

    public Room[] Rooms { get; set; }
}
```

The first thing to notice is that the name of each public property in the `Hotel` class will map to a field with the same name in the index definition. If you would like each field to start with a lower-case letter ("camel case"), you can tell the SDK to map the property names to camel-case automatically with the

`[SerializePropertyNamesAsCamelCase]` attribute on the class. This scenario is common in .NET applications that perform data-binding where the target schema is outside the control of the application developer without having to violate the "Pascal case" naming guidelines in .NET.

NOTE

The Azure Cognitive Search .NET SDK uses the [NewtonSoft JSON.NET](#) library to serialize and deserialize your custom model objects to and from JSON. You can customize this serialization if needed. For more information, see [Custom Serialization with JSON.NET](#).

The second thing to notice is each property is decorated with attributes such as `IsFilterable`, `IsSearchable`, `Key`, and `Analyzer`. 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.

The third important thing about the `Hotel` class is the data types of the public properties. The .NET types of these 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. The specific rules for the type mapping are documented with the `Documents.Get` method in the [Azure Cognitive Search .NET SDK reference](#). The `FieldBuilder` class takes care of this mapping for you, but it can still be helpful to understand in case you need to troubleshoot any serialization issues.

Did you happen to notice the `SmokingAllowed` property?

```
[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 does not allow smoking.

Some properties such as `Address` and `Rooms` are instances of .NET classes. These properties represent more complex data structures and, as a result, require fields with a [complex data type](#) in the index.

The `Address` property represents a set of multiple values in the `Address` class, defined below:

```
using System;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Newtonsoft.Json;

namespace AzureSearch.SDKHowTo
{
    public partial class Address
    {
        [IsSearchable]
        public string StreetAddress { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string City { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string StateProvince { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string PostalCode { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string Country { get; set; }
    }
}
```

This class contains the standard values used to describe addresses in the United States or Canada. You can use types like this to group logical fields together in the index.

The `Rooms` property represents an array of `Room` objects:

```

using System;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Newtonsoft.Json;

namespace AzureSearch.SDKHowTo
{
    public partial class Room
    {
        [IsSearchable]
        [Analyzer(AnalyzerNameAsString.EnMicrosoft)]
        public string Description { get; set; }

        [IsSearchable]
        [Analyzer(AnalyzerNameAsString.FrMicrosoft)]
        [JsonProperty("Description_fr")]
        public string DescriptionFr { get; set; }

        [IsSearchable, IsFilterable, IsFacetable]
        public string Type { get; set; }

        [IsFilterable, IsFacetable]
        public double? BaseRate { get; set; }

        [IsSearchable, IsFilterable, IsFacetable]
        public string BedOptions { get; set; }

        [IsFilterable, IsFacetable]
        public int SleepsCount { get; set; }

        [IsFilterable, IsFacetable]
        public bool? SmokingAllowed { get; set; }

        [IsSearchable, IsFilterable, IsFacetable]
        public string[] Tags { get; set; }
    }
}

```

Your data model in .NET and its corresponding index schema should be designed to support the search experience you'd like to give to your end user. Each top level object in .NET, ie document in the 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. We'll cover some query examples a little later.

This ability to use your own classes to interact with documents in the index works in both directions; You can also retrieve search results and have the SDK automatically deserialize them to a type of your choice, as we will see in the next section.

NOTE

The Azure Cognitive Search .NET SDK also supports dynamically-typed documents using the `Document` class, which is a key/value mapping of field names to field values. This is useful in scenarios where you don't know the index schema at design-time, or where it would be inconvenient to bind to specific model classes. All the methods in the SDK that deal with documents have overloads that work with the `Document` class, as well as strongly-typed overloads that take a generic type parameter. Only the latter are used in the sample code in this tutorial. The `Document` class inherits from `Dictionary<string, object>`.

Why you should use nullable data types

When designing your own model classes to map to an Azure Cognitive Search index, we recommend declaring properties of value types such as `bool` and `int` to be nullable (for example, `bool?` instead of `bool`). If you

use a non-nullable property, you have to **guarantee** that no documents in your index contain a null value for the corresponding field. Neither the SDK nor the Azure Cognitive Search service will help you to enforce this.

This is not just a hypothetical concern: Imagine a scenario where you add a new field to an existing index that is of type `Edm.Int32`. After updating the index definition, all documents will have a null value for that new field (since all types are nullable in Azure Cognitive Search). If you then use a model class with a non-nullable `int` property for that field, you will get a `JsonSerializationException` like this when trying to retrieve documents:

```
Error converting value {null} to type 'System.Int32'. Path 'IntValue'.
```

For this reason, we recommend that you use nullable types in your model classes as a best practice.

Custom Serialization with JSON.NET

The SDK uses JSON.NET for serializing and deserializing documents. You can customize serialization and deserialization if needed by defining your own `JsonConverter` or `IContractResolver`. For more information, see the [JSON.NET documentation](#). This can be useful when you want to adapt an existing model class from your application for use with Azure Cognitive Search, and other more advanced scenarios. For example, with custom serialization you can:

- Include or exclude certain properties of your model class from being stored as document fields.
- Map between property names in your code and field names in your index.
- Create custom attributes that can be used for mapping properties to document fields.

You can find examples of implementing custom serialization in the unit tests for the Azure Cognitive Search .NET SDK on GitHub. A good starting point is [this folder](#). It contains classes that are used by the custom serialization tests.

Searching for documents in the index

The last step in the sample application is to search for some documents in the index:

```

private static void RunQueries(ISearchIndexClient indexClient)
{
    SearchParameters parameters;
    DocumentSearchResult<Hotel> results;

    Console.WriteLine("Search the entire index for the term 'motel' and return only the HotelName field:\n");

    parameters =
        new SearchParameters()
    {
        Select = new[] { "HotelName" }
    };

    results = indexClient.Documents.Search<Hotel>("motel", parameters);

    WriteDocuments(results);

    Console.Write("Apply a filter to the index to find hotels with a room cheaper than $100 per night, ");
    Console.WriteLine("and return the hotelId and description:\n");

    parameters =
        new SearchParameters()
    {
        Filter = "Rooms/any(r: r/BaseRate lt 100)",
        Select = new[] { "HotelId", "Description" }
    };

    results = indexClient.Documents.Search<Hotel>("*", parameters);

    WriteDocuments(results);

    Console.Write("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");

    parameters =
        new SearchParameters()
    {
        OrderBy = new[] { "LastRenovationDate desc" },
        Select = new[] { "HotelName", "LastRenovationDate" },
        Top = 2
    };

    results = indexClient.Documents.Search<Hotel>("*", parameters);

    WriteDocuments(results);

    Console.WriteLine("Search the entire index for the term 'hotel':\n");

    parameters = new SearchParameters();
    results = indexClient.Documents.Search<Hotel>("hotel", parameters);

    WriteDocuments(results);
}

```

Each time it executes a query, this method first creates a new `SearchParameters` object. This object is used to specify additional options for the query such as sorting, filtering, paging, and facetting. In this method, we're setting the `Filter`, `Select`, `OrderBy`, and `Top` property for different queries. All the `SearchParameters` properties are documented [here](#).

The next step is to actually execute the search query. Running the search is done using the `Documents.Search` method. For each query, we pass the search text to use as a string (or `"*"` if there is no search text), plus the search parameters created earlier. We also specify `Hotel` as the type parameter for `Documents.Search`, which

tells the SDK to deserialize documents in the search results into objects of type `Hotel`.

NOTE

You can find more information about the search query expression syntax [here](#).

Finally, after each query this method iterates through all the matches in the search results, printing each document to the console:

```
private static void WriteDocuments(DocumentSearchResult<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.Results)
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}
```

Let's take a closer look at each of the queries in turn. Here is the code to execute the first query:

```
parameters =
    new SearchParameters()
    {
        Select = new[] { "HotelName" }
    };

results = indexClient.Documents.Search<Hotel>("motel", parameters);

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` parameter. Here are the results:

```
Name: Secret Point Motel

Name: Twin Dome Motel
```

The next query is a little more interesting. We want to find any hotels that have a room with a nightly rate of less than \$100 and return only the hotel ID and description:

```
parameters =
    new SearchParameters()
    {
        Filter = "Rooms/any(r: r/BaseRate lt 100)",
        Select = new[] { "HotelId", "Description" }
    };

results = indexClient.Documents.Search<Hotel>("", parameters);

WriteDocuments(results);
```

This 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. You can find out more about the OData syntax that Azure Cognitive Search supports [here](#).

Here are the results of the query:

```
HotelId: 1
Description: The hotel is ideally located on the main commercial artery of the city in the heart of New York...
```

```
HotelId: 2
Description: The hotel is situated in a nineteenth century plaza, which has been expanded and renovated to...
```

Next, we want to find the top two hotels that have been most recently renovated, and show the hotel name and last renovation date. Here is the code:

```
parameters =
    new SearchParameters()
{
    OrderBy = new[] { "LastRenovationDate desc" },
    Select = new[] { "HotelName", "LastRenovationDate" },
    Top = 2
};

results = indexClient.Documents.Search<Hotel>("*", parameters);

WriteDocuments(results);
```

In this case, we again use OData syntax to specify the `OrderBy` parameter as `lastRenovationDate desc`. We also set `Top` to 2 to ensure we only get the top two documents. As before, we set `Select` to specify which fields should be returned.

Here are the results:

```
Name: Fancy Stay      Last renovated on: 6/27/2010 12:00:00 AM +00:00
Name: Roach Motel      Last renovated on: 4/28/1982 12:00:00 AM +00:00
```

Finally, we want to find all hotels names that match the word "hotel":

```
parameters = new SearchParameters()
{
    SearchFields = new[] { "HotelName" }
};
results = indexClient.Documents.Search<Hotel>("hotel", parameters);

WriteDocuments(results);
```

And here are the results, which include all fields since we did not specify the `Select` property:

```
HotelId: 3
Name: Triple Landscape Hotel
...
```

This step completes the tutorial, but don't stop here. **Next steps provide additional resources for learning more about Azure Cognitive Search.

Next steps

- Browse the references for the [.NET SDK](#) and [REST API](#).
- Review [naming conventions](#) to learn the rules for naming various objects.
- Review [supported data types](#) in Azure Cognitive Search.

Quickstart: Create a search index using the Microsoft.Azure.Search v10 client library

10/4/2020 • 17 minutes to read • [Edit Online](#)

This article is the C# quickstart for the legacy Microsoft.Azure.Search (version 10) client library, now superseded by the Azure.Search.Documents (version 11) client library. If you have existing search solutions that uses the Microsoft.Azure.Search libraries, you can use this quickstart to learn about those APIs.

For new solutions, we recommend the new Azure.Search.Documents library. For an introduction, see [Quickstart: Create a search index using Azure.Search.Documents library](#).

About this quickstart

Create a .NET Core console application in C# that creates, loads, and queries an Azure Cognitive Search index using Visual Studio and the [Microsoft.Azure.Search client libraries](#).

This article explains how to create the application. You could also [download and run the complete application](#).

NOTE

The demo code in this article uses the synchronous methods of the Azure Cognitive Search version 10 .NET SDK for simplicity. However, for production scenarios, we recommend using the asynchronous methods in your own applications to keep them scalable and responsive. For example, you could use `CreateAsync` and `DeleteAsync` instead of `Create` and `Delete`.

Prerequisites

Before you begin, you must have the following:

- 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.
- [Visual Studio](#), any edition. Sample code and instructions were tested on the free Community edition.

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.

Get the query key as well. It's a best practice to issue query requests with read-only access.

The screenshot shows the Azure portal interface for managing an Azure Cognitive Search service named 'mydemo'. The main page displays basic service information like the resource group ('demo-resource-group') and a placeholder URL. The 'Keys' tab is selected in the sub-navigation menu, which is highlighted with a red box. Within the 'Keys' section, the 'Primary admin key' field is also highlighted with a red box and contains a placeholder string. Red circles with numbers 1 and 2 indicate specific points of interest: 1 is on the URL, and 2 is on the 'Keys' tab.

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.

Set up your environment

Begin by opening Visual Studio and creating a new Console App project that can run on .NET Core.

Install NuGet packages

The [Microsoft.Azure.Search package](#) consists of a few client libraries that are distributed as NuGet packages.

For this project, use version 10 of the `Microsoft.Azure.Search` NuGet package and the latest `Microsoft.Extensions.Configuration.Json` NuGet package.

1. In Tools > NuGet Package Manager, select Manage NuGet Packages for Solution....
2. Click Browse.
3. Search for `Microsoft.Azure.Search` and select version 10.
4. Click Install on the right to add the assembly to your project and solution.
5. Repeat for `Microsoft.Extensions.Configuration.Json`, selecting version 2.2.0 or later.

Add Azure Cognitive Search service information

1. In Solution Explorer, right click on the project and select Add > New Item... .
2. In Add New Item, search for "JSON" to return a JSON-related list of item types.
3. Choose **JSON File**, name the file "appsettings.json", and click **Add**.
4. Add the file to your output directory. Right-click `appsettings.json` and select **Properties**. In **Copy to Output Directory**, select **Copy if newer**.
5. Copy the following JSON into your new JSON file.

```
{
  "SearchServiceName": "<YOUR-SEARCH-SERVICE-NAME>",
  "SearchServiceAdminApiKey": "<YOUR-ADMIN-API-KEY>",
  "SearchIndexName": "hotels-quickstart"
}
```

- Replace the search service name (YOUR-SEARCH-SERVICE-NAME) and admin API key (YOUR-ADMIN-API-KEY) with valid values. If your service endpoint is <https://mydemo.search.windows.net>, the service name would be "mydemo".

Add class ".Method" files to your project

This step is required to produce meaningful output in the console. When printing results to the console window, individual fields from the Hotel object must be returned as strings. This step implements `ToString()` to perform this task, which you do by copying the necessary code to two new files.

- Add two empty class definitions to your project: Address.Methods.cs, Hotel.Methods.cs
- In Address.Methods.cs, overwrite the default contents with the following code, [lines 1-25](#).
- In Hotel.Methods.cs, copy [lines 1-68](#).

1 - Create index

The hotels index consists of simple and complex fields, where a simple field is "HotelName" or "Description", and complex fields are an address with subfields, or a collection of rooms. When an index includes complex types, isolate the complex field definitions in separate classes.

- Add two empty class definitions to your project: Address.cs, Hotel.cs
- In Address.cs, overwrite the default contents with the following code:

```
using System;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Newtonsoft.Json;

namespace AzureSearchQuickstart
{
    public partial class Address
    {
        [IsSearchable]
        public string StreetAddress { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string City { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string StateProvince { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string PostalCode { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string Country { get; set; }
    }
}
```

- In Hotel.cs, the class defines the overall structure of the index, including references to the address class.

```

namespace AzureSearchQuickstart
{
    using System;
    using Microsoft.Azure.Search;
    using Microsoft.Azure.Search.Models;
    using Newtonsoft.Json;

    public partial class Hotel
    {
        [System.ComponentModel.DataAnnotations.Key]
        [IsFilterable]
        public string HotelId { get; set; }

        [IsSearchable, IsSortable]
        public string HotelName { get; set; }

        [IsSearchable]
        [Analyzer(AnalyzerNameAsString.EnMicrosoft)]
        public string Description { get; set; }

        [IsSearchable]
        [Analyzer(AnalyzerNameAsString.FrLucene)]
        [JsonProperty("Description_fr")]
        public string DescriptionFr { get; set; }

        [IsSearchable, IsFilterable, IsSortable, IsFacetable]
        public string Category { get; set; }

        [IsSearchable, IsFilterable, IsFacetable]
        public string[] Tags { get; set; }

        [IsFilterable, IsSortable, IsFacetable]
        public bool? ParkingIncluded { get; set; }

        [IsFilterable, IsSortable, IsFacetable]
        public DateTimeOffset? LastRenovationDate { get; set; }

        [IsFilterable, IsSortable, IsFacetable]
        public double? Rating { get; set; }

        public Address Address { get; set; }
    }
}

```

Attributes on the field determine how it is used in an application. For example, the `IsSearchable` attribute must be assigned to every field that should be included in a full text search.

NOTE

In the .NET SDK, fields must be explicitly attributed as `IsSearchable`, `IsFilterable`, `IsSortable`, and `IsFacetable`. This behavior is in contrast with the REST API which implicitly enables attribution based on data type (for example, simple string fields are automatically searchable).

Exactly one field in your index of type `string` must be the *key* field, uniquely identifying each document. In this schema, the key is `HotelId`.

In this index, the description fields use the optional `analyzer` property, specified when you want to override the default standard Lucene analyzer. The `description_fr` field is using the French Lucene analyzer (`FrLucene`) because it stores French text. The `description` is using the optional Microsoft language analyzer (`EnMicrosoft`).

4. In Program.cs, create an instance of the `SearchServiceClient` class to connect to the service, using values

that are stored in the application's config file (appsettings.json).

`SearchServiceClient` has an `Indexes` property, providing all the methods you need to create, list, update, or delete Azure Cognitive Search indexes.

```
using System;
using System.Linq;
using System.Threading;
using Microsoft.Azure.Search;
using Microsoft.Azure.Search.Models;
using Microsoft.Extensions.Configuration;

namespace AzureSearchQuickstart
{
    class Program {
        // Demonstrates index delete, create, load, and query
        // Commented-out code is uncommented in later steps
        static void Main(string[] args)
        {
            IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.json");
            IConfigurationRoot configuration = builder.Build();

            SearchServiceClient serviceClient = CreateSearchServiceClient(configuration);

            string indexName = configuration["SearchIndexName"];

            Console.WriteLine("{0}", "Deleting index...\n");
            DeleteIndexIfExists(indexName, serviceClient);

            Console.WriteLine("{0}", "Creating index...\n");
            CreateIndex(indexName, serviceClient);

            // Uncomment next 3 lines in "2 - Load documents"
            // ISearchIndexClient indexClient = serviceClient.Indexes.GetClient(indexName);
            // Console.WriteLine("{0}", "Uploading documents...\n");
            // UploadDocuments(indexClient);

            // Uncomment next 2 lines in "3 - Search an index"
            // Console.WriteLine("{0}", "Searching index...\n");
            // RunQueries(indexClient);

            Console.WriteLine("{0}", "Complete. Press any key to end application...\n");
            Console.ReadKey();
        }

        // Create the search service client
        private static SearchServiceClient CreateSearchServiceClient(IConfigurationRoot configuration)
        {
            string searchServiceName = configuration["SearchServiceName"];
            string adminApiKey = configuration["SearchServiceAdminApiKey"];

            SearchServiceClient serviceClient = new SearchServiceClient(searchServiceName, new
SearchCredentials(adminApiKey));
            return serviceClient;
        }

        // Delete an existing index to reuse its name
        private static void DeleteIndexIfExists(string indexName, SearchServiceClient serviceClient)
        {
            if (serviceClient.Indexes.Exists(indexName))
            {
                serviceClient.Indexes.Delete(indexName);
            }
        }

        // Create an index whose fields correspond to the properties of the Hotel class.
        // The Address property of Hotel will be modeled as a complex field.
        // The Address property of Hotel will be modeled as a complex field.
    }
}
```

```

    // The properties of the Address class in turn correspond to sub-fields of the Address complex
    field.

    // The fields of the index are defined by calling the FieldBuilder.BuildForType() method.
    private static void CreateIndex(string indexName, SearchServiceClient serviceClient)
    {
        var definition = new Microsoft.Azure.Search.Models.Index()
        {
            Name = indexName,
            Fields = FieldBuilder.BuildForType<Hotel>()
        };

        serviceClient.Indexes.Create(definition);
    }
}
}

```

If possible, share a single instance of `SearchServiceClient` in your application to avoid opening too many connections. Class methods are thread-safe to enable such sharing.

The class has several constructors. The one you want takes your search service name and a `SearchCredentials` object as parameters. `SearchCredentials` wraps your api-key.

In the index definition, the easiest way to create the `Field` objects is by calling the `FieldBuilder.BuildForType` method, passing a model class for the type parameter. A model class has properties that map to the fields of your index. This mapping allows you to bind documents from your search index to instances of your model class.

NOTE

If you don't plan to use a model class, you can still define your index by creating `Field` objects directly. You can provide the name of the field to the constructor, along with the data type (or analyzer for string fields). You can also set other properties like `IsSearchable`, `IsFilterable`, to name a few.

5. Press F5 to build the app and create the index.

If the project builds successfully, a console window opens, writing status messages to the screen for deleting and creating the index.

2 - Load documents

In Azure Cognitive 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 `IndexBatch` object. An `IndexBatch` contains a collection of `IndexAction` objects, 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 `IndexBatch`. The documents below conform to the hotel-quickstart index, as defined by the hotel and address classes.

```

// Upload documents as a batch
private static void UploadDocuments(ISearchIndexClient indexClient)
{
    var actions = new IndexAction<Hotel>[]
    {
        IndexAction.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, TimeSpan.Zero),
            Rating = 3.6,
            Address = new Address()
            {
                StreetAddress = "677 5th Ave",
                City = "New York",
                StateProvince = "NY",
                PostalCode = "10022",
                Country = "USA"
            }
        },
        IndexAction.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"
                }
            }
        ),
        IndexAction.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"
        }
    }
),
IndexAction.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, TimeSpan.Zero),
        Rating = 4.6,
        Address = new Address()
        {
            StreetAddress = "7400 San Pedro Ave",
            City = "San Antonio",
            StateProvince = "TX",
            PostalCode = "78216",
            Country = "USA"
        }
    }
),
);
};

var batch = IndexBatch.New(actions);

try
{
    indexClient.Documents.Index(batch);
}
catch (IndexBatchException e)
{
    // When a service is under load, indexing might fail for some documents in the batch.
    // Depending on your application, you can compensate by 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}",
        String.Join(", ", e.IndexingResults.Where(r => !r.Succeeded).Select(r => r.Key)));
}

// Wait 2 seconds before starting queries
Console.WriteLine("Waiting for indexing...\n");
Thread.Sleep(2000);
}

```

Once you initialize the `IndexBatch` object, you can send it to the index by calling `Documents.Index` on your `SearchIndexClient` object. `Documents` is a property of `SearchIndexClient` that provides methods for adding, modifying, deleting, or querying documents in your index.

The `try / catch` surrounding the call to the `Index` method catches indexing failures, which might happen if your service is under heavy load. In production code, you could delay and then retry indexing the

documents that failed, or log and continue like the sample does, or handle it in some other way that meets your application's data consistency requirements.

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.

2. In Program.cs, in main, uncomment the lines for "2 - Load documents".

```
// Uncomment next 3 lines in "2 - Load documents"
ISearchIndexClient indexClient = serviceClient.Indexes.GetClient(indexName);
Console.WriteLine("{0}", "Uploading documents...\n");
UploadDocuments(indexClient);
```

3. Press F5 to rebuild the app.

If the project builds successfully, a console window opens, writing status messages, this time with a message about uploading documents. In the Azure portal, in the search service **Overview** page, the hotels-quickstart index should now have 4 documents.

For more information about document processing, see ["How the .NET SDK handles documents"](#).

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 as well as other [parameters](#).

The [DocumentSearchResult](#) class represents the results.

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

```
private static void WriteDocuments(DocumentSearchResult<Hotel> searchResults)
{
    foreach (SearchResult<Hotel> result in searchResults.Results)
    {
        Console.WriteLine(result.Document);
    }

    Console.WriteLine();
}
```

2. Create a RunQueries method to execute queries and return results. Results are Hotel objects. You can use the select parameter to surface individual fields. If a field is not included in the select parameter, its corresponding Hotel property will be null.

```

private static void RunQueries(ISearchIndexClient indexClient)
{
    SearchParameters parameters;
    DocumentSearchResult<Hotel> results;

    // Query 1
    Console.WriteLine("Query 1: Search for term 'Atlanta' with no result trimming");
    parameters = new SearchParameters();
    results = indexClient.Documents.Search<Hotel>("Atlanta", parameters);
    WriteDocuments(results);

    // Query 2
    Console.WriteLine("Query 2: Search on the term 'Atlanta', with trimming");
    Console.WriteLine("Returning only these fields: HotelName, Tags, Address:\n");
    parameters =
        new SearchParameters()
    {
        Select = new[] { "HotelName", "Tags", "Address" },
    };
    results = indexClient.Documents.Search<Hotel>("Atlanta", parameters);
    WriteDocuments(results);

    // Query 3
    Console.WriteLine("Query 3: Search for the terms 'restaurant' and 'wifi'");
    Console.WriteLine("Return only these fields: HotelName, Description, and Tags:\n");
    parameters =
        new SearchParameters()
    {
        Select = new[] { "HotelName", "Description", "Tags" }
    };
    results = indexClient.Documents.Search<Hotel>("restaurant, wifi", parameters);
    WriteDocuments(results);

    // Query 4 -filtered query
    Console.WriteLine("Query 4: Filter on ratings greater than 4");
    Console.WriteLine("Returning only these fields: HotelName, Rating:\n");
    parameters =
        new SearchParameters()
    {
        Filter = "Rating gt 4",
        Select = new[] { "HotelName", "Rating" }
    };
    results = indexClient.Documents.Search<Hotel>("*", parameters);
    WriteDocuments(results);

    // Query 5 - top 2 results
    Console.WriteLine("Query 5: Search on term 'boutique'");
    Console.WriteLine("Sort by rating in descending order, taking the top two results");
    Console.WriteLine("Returning only these fields: HotelId, HotelName, Category, Rating:\n");
    parameters =
        new SearchParameters()
    {
        OrderBy = new[] { "Rating desc" },
        Select = new[] { "HotelId", "HotelName", "Category", "Rating" },
        Top = 2
    };
    results = indexClient.Documents.Search<Hotel>("boutique", parameters);
    WriteDocuments(results);
}

```

There are two ways of matching terms in a query: full-text search, and filters. A full-text search query searches for one or more terms in `IsSearchable` fields in your index. A filter is a boolean expression that is evaluated over `IsFilterable` fields in an index. You can use full-text search and filters together or separately.

Both searches and filters are performed using the `Documents.Search` method. A search query can be passed

in the `searchText` parameter, while a filter expression can be passed in the `Filter` property of the `SearchParameters` class. To filter without searching, just pass `"*"` for the `searchText` parameter. To search without filtering, just leave the `Filter` property unset, or do not pass in a `SearchParameters` instance at all.

3. In Program.cs, in main, uncomment the lines for "3 - Search".

```
// Uncomment next 2 lines in "3 - Search an index"
Console.WriteLine("{0}", "Searching documents...\n");
RunQueries(indexClient);
```

4. The solution is now finished. Press F5 to rebuild the app and run the program in its entirety.

Output includes the same messages as before, with 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 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 C# quickstart, you worked through a series 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. If you are comfortable with the basic concepts, we recommend the next article for an exploration of alternative approaches and concepts that will deepen your knowledge.

The sample code and index are expanded versions of this one. The next sample adds a Rooms collection, uses different classes and actions, and takes a closer look at how processing works.

[How to develop in .NET](#)

Want to optimize and save on your cloud spending?

[Start analyzing costs with Cost Management](#)

Upgrade to Azure Cognitive Search .NET SDK version 11

10/4/2020 • 6 minutes to read • [Edit Online](#)

If you're using version 10.0 or older of the [.NET SDK](#), this article will help you upgrade to version 11.

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). The library has been redesigned for greater consistency with other Azure client libraries, taking a dependency on [Azure.Core](#) and [System.Text.Json](#), and implementing familiar approaches for common tasks.

Some key differences you'll notice in the new version include:

- One package and library as opposed to multiple
- A new package name: `Azure.Search.Documents` instead of `Microsoft.Azure.Search`.
- Three clients instead of two: `SearchClient`, `SearchIndexClient`, `SearchIndexerClient`
- Naming differences across a range of APIs and small structural differences that simplify some tasks

NOTE

Review the [change log](#) for an itemized list of changes in .NET SDK version 11.

Package and library consolidation

Version 11 consolidates multiple packages and libraries into one. Post-migration, you will have fewer libraries to manage.

- [Azure.Search.Documents package](#)
- [API reference for the client library](#)

Client differences

Where applicable, the following table maps the client libraries between the two versions.

SCOPE OF OPERATIONS	MICROSOFT.AZURE.SEARCH (V10)	AZURE.SEARCH.DOCUMENTS (V11)
Client used for queries and to populate an index.	<code>SearchIndexClient</code>	<code>SearchClient</code>
Client used for indexes, analyzers, synonym maps	<code>SearchServiceClient</code>	<code>SearchIndexClient</code>
Client used for indexers, data sources, skillsets	<code>SearchServiceClient</code>	<code>SearchIndexerClient (new)</code>

IMPORTANT

`SearchIndexClient` exists in both versions, but supports different things. In version 10, `SearchIndexClient` creates indexes and other objects. In version 11, `SearchIndexClient` works with existing indexes. 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 is not 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
<code>SearchCredentials</code>	<code>AzureKeyCredential</code>
<code>EncryptionKey</code> (existed in the preview SDK as a generally available feature)	<code>SearchResourceEncryptionKey</code>

Indexes, analyzers, synonym maps

VERSION 10	VERSION 11 EQUIVALENT
<code>Index</code>	<code>SearchIndex</code>
<code>Field</code>	<code>SearchField</code>
<code>DataType</code>	<code>SearchFieldDataType</code>
<code>ItemError</code>	<code>SearchIndexerError</code>
<code>Analyzer</code>	<code>LexicalAnalyzer</code> (also, <code>AnalyzerName</code> to <code>LexicalAnalyzerName</code>)
<code>AnalyzeRequest</code>	<code>AnalyzeTextOptions</code>
<code>StandardAnalyzer</code>	<code>LuceneStandardAnalyzer</code>
<code>StandardTokenizer</code>	<code>LuceneStandardTokenizer</code> (also, <code>StandardTokenizerV2</code> to <code>LuceneStandardTokenizerV2</code>)
<code>TokenInfo</code>	<code>AnalyzedTokenInfo</code>
<code>Tokenizer</code>	<code>LexicalTokenizer</code> (also, <code>TokenizerName</code> to <code>LexicalTokenizerName</code>)
<code>SynonymMap.Format</code>	None. Remove references to <code>Format</code> .

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

Query definitions and results

VERSION 10	VERSION 11 EQUIVALENT
DocumentSearchResult	SearchResult or SearchResults, depending on whether the result is a single document or multiple.
DocumentSuggestResult	SuggestResults
SearchParameters	SearchOptions

What's in version 11

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 [REST API documentation](#) if you want more background on specific objects or operations.

Version 11 targets the [2020-06-30 search service](#). Because version 11 is also a new client library built from the ground up, most of the development effort has focused on equivalency with version 10, with some REST API feature support still pending.

Version 11.0 fully supports the following objects and operations:

- Index creation and management
- Synonym map creation and management
- All query types and syntax (except geo-spatial filters)
- Indexer objects and operations for indexing Azure data sources, including data sources and skillsets

Version 11.1 adds the following:

- [FieldBuilder](#) (added in 11.1)
- [Serializer property](#) (added in 11.1) to support custom serialization

Pending features

The following version 10 features are not yet available in version 11. If you require these features, hold off on migration until they are supported.

- geospatial types
- [Knowledge store](#)

Steps to upgrade

The following steps get you started on a code migration by walking through the first set of required tasks, especially in regards 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:

```
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:

```
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 are using indexers, datasources, or skillsets, change the client references to [SearchIndexerClient](#). This client is new in version 11 and has no antecedent.
6. 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.
7. Update client references for index, indexer, synonym map, and analyzer objects. Instances of [SearchServiceClient](#) should be changed to [SearchIndexClient](#).
8. As much as possible, 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.

9. 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 in version 11

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 will 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.

Next steps

- [Azure.Search.Documents package](#)
- [Samples on GitHub](#)
- [Azure.Search.Document API reference](#)

Upgrade to Azure Cognitive Search .NET SDK version 10

10/4/2020 • 6 minutes to read • [Edit Online](#)

If you're using version 9.0 or older of the [.NET SDK](#), this article will help you upgrade your application to use version 10.

Azure Search is renamed to Azure Cognitive Search in version 10, but namespaces and package names are unchanged. Previous versions of the SDK (9.0 and earlier) continue to use the former name. For more information about using the SDK, including examples, see [How to use Azure Cognitive Search from a .NET Application](#).

Version 10 adds several features and bug fixes, bringing it to the same functional level as the REST API version [2019-05-06](#). In cases where a change breaks existing code, we'll walk you through the [steps required to resolve the issue](#).

NOTE

If you're using version 8.0-preview or older, you should upgrade to version 9 first, and then upgrade to version 10. See [Upgrading to the Azure Search .NET SDK version 9](#) for instructions.

Your search service instance supports several REST API versions, including the latest one. You can continue to use a version when it is no longer the latest one, but we recommend that you migrate your code to use the newest version. When using the REST API, you must specify the API version in every request via the `api-version` parameter. When using the .NET SDK, the version of the SDK you're using determines the corresponding version of the REST API. If you are using an older SDK, you can continue to run that code with no changes even if the service is upgraded to support a newer API version.

What's new in version 10

Version 10 of the Azure Cognitive Search .NET SDK targets REST API [2019-05-06](#) with these updates:

- Introduction of two new skills - [Conditional skill](#) and [Text Translation skill](#).
- [Shaper skill](#) inputs have been restructured to accommodate consolidation from nested contexts. For more information, see this [example JSON definition](#).
- Addition of two new [field mapping functions](#):
 - [urlEncode](#)
 - [urlDecode](#)
- On certain occasions, errors and warnings that show up in [indexer execution status](#) can have additional details that help in debugging. [IndexerExecutionResult](#) has been updated to reflect this behavior.
- Individual skills defined within a [skillset](#) can optionally be identified by specifying a [name](#) property.
- [ServiceLimits](#) shows limits for [complex types](#) and [IndexerExecutionInfo](#) shows pertinent indexer limits/quotas.

Steps to upgrade

1. Update your NuGet reference for [Microsoft.Azure.Search](#) using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.
2. Once NuGet has downloaded the new packages and their dependencies, rebuild your project.
3. If your build fails, you will need to fix each build error. See [Breaking changes in version 10](#) for details on how to resolve each potential build error.

- Once you've fixed any build errors or warnings, you can make changes to your application to take advantage of new functionality if you wish. New features in the SDK are detailed in [What's new in version 10](#).

Breaking changes in version 10

There are several breaking changes in version 10 that may require code changes in addition to rebuilding your application.

NOTE

The list of changes below is not exhaustive. Some changes will likely not result in build errors, but are technically breaking since they break binary compatibility with assemblies that depend on earlier versions of the Azure Cognitive Search .NET SDK assemblies. Significant changes that fall under this category are also listed along with recommendations. Please rebuild your application when upgrading to version 10 to avoid any binary compatibility issues.

Custom Web API skill definition

The definition of the [Custom Web API skill](#) was incorrectly specified in version 9 and older.

The model for `WebApiSkill` specified `HttpHeaders` as an object property that *contains* a dictionary. Creating a skillset with a `WebApiSkill` constructed in this manner would result in an exception because the REST API would consider the request badly formed. This issue has been corrected, by making `HttpHeaders` a top-level dictionary property on the `WebApiSkill` model itself - which is considered a valid request from the REST API.

For example, if you previously attempted to instantiate a `WebApiSkill` as follows:

```
var webApiSkill = new WebApiSkill(
    inputs,
    outputs,
    uri: "https://contoso.example.org")
{
    HttpHeaders = new WebApiHttpHeaders()
    {
        Headers = new Dictionary<string, string>()
        {
            ["header"] = "value"
        }
    }
};
```

change it to the following, to avoid the validation error from the REST API:

```
var webApiSkill = new WebApiSkill(
    inputs,
    outputs,
    uri: "https://contoso.example.org")
{
    HttpHeaders = new Dictionary<string, string>()
    {
        ["header"] = "value"
    }
};
```

Shaper skill allows nested context consolidation

Shaper skill can now allow input consolidation from nested contexts. To enable this change, we modified `InputFieldMappingEntry` so that it can be instantiated by specifying just a `Source` property, or both the `SourceContext` and `Inputs` properties.

You will most likely not need to make any code changes; however note that only one of these two combinations is allowed. This means:

- Creating an `InputFieldMappingEntry` where only `Source` is initialized is valid.
- Creating an `InputFieldMappingEntry` where only `SourceContext` and `Inputs` are initialized is valid.
- All other combinations involving those three properties are invalid.

If you decide to start making use of this new capability, make sure all your clients are updated to use version 10 first, before rolling out that change. Otherwise, there is a possibility that an update by a client (using an older version of the SDK) to the Shaper skill may result in validation errors.

NOTE

Even though the underlying `InputFieldMappingEntry` model has been modified to allow consolidation from nested contexts, its use is only valid within the definition of a Shaper skill. Using this capability in other skills, while valid at compile time, will result in a validation error at runtime.

Skills can be identified by a name

Each skill within a skillset now has a new property `Name`, which can be initialized in your code to help identify the skill. This is optional - when unspecified (which is the default, if no explicit code change was made), it is assigned a default name using the 1-based index of the skill in the skillset, prefixed with the '#' character. For example, in the following skillset definition (most initializations skipped for brevity):

```
var skillset = new Skillset()
{
    Skills = new List<Skill>()
    {
        new SentimentSkill(),
        new WebApiSkill(),
        new ShaperSkill(),
        ...
    }
}
```

`SentimentSkill` is assigned a name `#1`, `WebApiSkill` is assigned `#2`, `ShaperSkill` is assigned `#3` and so on.

If you choose to identify skills by a custom name, make sure to update all instances of your clients to version 10 of the SDK first. Otherwise, there is a possibility that a client using an older version of the SDK could `null` out the `Name` property of a skill, causing the client to fall back on the default naming scheme.

Details about errors and warnings

`ItemError` and `ItemWarning` models that encapsulate details of errors and warnings (respectively) that occur during an indexer execution have been modified to include three new properties with the objective to aid in debugging the indexer. These properties are:

- `Name` : The name of the source at which the error originated. For example, it could refer to a particular skill in the attached skillset.
- `Details` : Additional verbose details about the error or warning.
- `DocumentationLink` : A link to a troubleshooting guide for the specific error or warning.

NOTE

We have started to structure our errors and warnings to include these useful details whenever possible. We are working to make sure that for all errors and warnings these details are present, but it is a work in progress and these additional details may not always be populated.

Next steps

- Changes to the Shaper skill have the most potential impact on new or existing code. As a next step, be sure to revisit this example illustrating the input structure: [Shaper skill JSON definition example](#)
- Go through the [AI enrichment overview](#).
- We welcome your feedback on the SDK. If you encounter problems, feel free to ask us for help on [Stack Overflow](#). If you find a bug, you can file an issue in the [Azure .NET SDK GitHub repository](#). Make sure to prefix your issue title with "[Azure Cognitive Search]".

Upgrade to Azure Search .NET SDK version 9

10/4/2020 • 7 minutes to read • [Edit Online](#)

If you're using version 7.0-preview or older of the [Azure Search .NET SDK](#), this article will help you upgrade your application to use version 9.

NOTE

If you wish to use version 8.0-preview to evaluate features that are not generally available yet, you can also follow the instructions in this article to upgrade to 8.0-preview from prior versions.

For a more general walkthrough of the SDK including examples, see [How to use Azure Search from a .NET Application](#).

Version 9 of the Azure Search .NET SDK contains many changes from earlier versions. Some of these are breaking changes, but they should only require relatively minor changes to your code. See [Steps to upgrade](#) for instructions on how to change your code to use the new SDK version.

NOTE

If you're using version 4.0-preview or older, you should upgrade to version 5 first, and then upgrade to version 9. See [Upgrading to the Azure Search .NET SDK version 5](#) for instructions.

Your Azure Search service instance supports several REST API versions, including the latest one. You can continue to use a version when it is no longer the latest one, but we recommend that you migrate your code to use the newest version. When using the REST API, you must specify the API version in every request via the `api-version` parameter. When using the .NET SDK, the version of the SDK you're using determines the corresponding version of the REST API. If you are using an older SDK, you can continue to run that code with no changes even if the service is upgraded to support a newer API version.

What's new in version 9

Version 9 of the Azure Search .NET SDK targets 2019-05-06 version of Azure Search REST API, with the following features:

- [AI enrichment](#) is the ability to extract text from images, blobs, and other unstructured data sources - enriching the content to make it more searchable in an Azure Search index.
- Support for [complex types](#) allows you to model almost any nested JSON structure in an Azure Search index.
- [Autocomplete](#) provides an alternative to the [Suggest](#) API for implementing search-as-you-type behavior. Autocomplete "finishes" the word or phrase that a user is currently typing.
- [JsonLines parsing mode](#), part of Azure Blob indexing, creates one search document per JSON entity that is separated by a newline.

New preview features in version 8.0-preview

Version 8.0-preview of the Azure Search .NET SDK targets API version 2017-11-11-Preview. This version includes all the same features of version 9, plus:

- [Customer-managed encryption keys](#) for service-side encryption-at-rest is a new preview feature. In addition to the built-in encryption-at-rest managed by Microsoft, you can apply an additional layer of encryption where you are the sole owner of the keys.

Steps to upgrade

First, update your NuGet reference for `Microsoft.Azure.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

Once NuGet has downloaded the new packages and their dependencies, rebuild your project. Depending on how your code is structured, it may rebuild successfully. If so, you're ready to go!

If your build fails, you will need to fix each build error. See [Breaking changes in version 9](#) for details on how to resolve each potential build error.

You may see additional build warnings related to obsolete methods or properties. The warnings will include instructions on what to use instead of the deprecated feature. For example, if your application uses the `DataSourceType.DocumentDb` property, you should get a warning that says "This member is deprecated. Use `CosmosDb` instead".

Once you've fixed any build errors or warnings, you can make changes to your application to take advantage of new functionality if you wish. New features in the SDK are detailed in [What's new in version 9](#).

Breaking changes in version 9

There are several breaking changes in version 9 that may require code changes in addition to rebuilding your application.

NOTE

The list of changes below is not exhaustive. Some changes will likely not result in build errors, but are technically breaking since they break binary compatibility with assemblies that depend on earlier versions of the Azure Search .NET SDK assemblies. Such changes are not listed below. Please rebuild your application when upgrading to version 9 to avoid any binary compatibility issues.

Immutable properties

The public properties of several model classes are now immutable. If you need to create custom instances of these classes for testing, you can use the new parameterized constructors:

- `AutocompleteItem`
- `DocumentSearchResult`
- `DocumentSuggestResult`
- `FacetResult`
- `SearchResult`
- `SuggestResult`

Changes to Field

The `Field` class has changed now that it can also represent complex fields.

The following `bool` properties are now nullable:

- `IsFilterable`
- `IsFacetable`
- `IsSearchable`
- `IsSortable`
- `IsRetrievable`
- `IsKey`

This is because these properties must now be `null` in the case of complex fields. If you have code that reads these properties, it has to be prepared to handle `null`. Note that all other properties of `Field` have always been and continue to be nullable, and some of those will also be `null` in the case of complex fields -- specifically the following:

- `Analyzer`
- `SearchAnalyzer`
- `IndexAnalyzer`
- `SynonymMaps`

The parameterless constructor of `Field` has been made `internal`. From now on, every `Field` requires an explicit name and data type at the time of construction.

Simplified batch and results types

In version 7.0-preview and earlier, the various classes that encapsulate groups of documents were structured into parallel class hierarchies:

- `DocumentSearchResult` and `DocumentSearchResult<T>` inherited from `DocumentSearchResultBase`
- `DocumentSuggestResult` and `DocumentSuggestResult<T>` inherited from `DocumentSuggestResultBase`
- `IndexAction` and `IndexAction<T>` inherited from `IndexActionBase`
- `IndexBatch` and `IndexBatch<T>` inherited from `IndexBatchBase`
- `SearchResult` and `SearchResult<T>` inherited from `SearchResultBase`
- `SuggestResult` and `SuggestResult<T>` inherited from `SuggestResultBase`

The derived types without a generic type parameter were meant to be used in "dynamically-typed" scenarios and assumed usage of the `Document` type.

Starting with version 8.0-preview, the base classes and non-generic derived classes have all been removed. For dynamically-typed scenarios, you can use `IndexBatch<Document>`, `DocumentSearchResult<Document>`, and so on.

Removed ExtensibleEnum

The `ExtensibleEnum` base class has been removed. All classes that derived from it are now structs, such as `AnalyzerName`, `DataType`, and `DataSourceType` for example. Their `Create` methods have also been removed. You can just remove calls to `Create` since these types are implicitly convertible from strings. If that results in compiler errors, you can explicitly invoke the conversion operator via casting to disambiguate types. For example, you can change code like this:

```
var index = new Index()
{
    Fields = new[]
    {
        new Field("id", DataType.String) { IsKey = true },
        new Field("message", AnalyzerName.Create("my_email_analyzer")) { IsSearchable = true }
    },
    ...
}
```

to this:

```
var index = new Index()
{
    Fields = new[]
    {
        new Field("id", DataType.String) { IsKey = true },
        new Field("message", (AnalyzerName)"my_email_analyzer") { IsSearchable = true }
    },
    ...
}
```

Properties that held optional values of these types are now explicitly typed as nullable so they continue to be optional.

Removed FacetResults and HitHighlights

The `FacetResults` and `HitHighlights` classes have been removed. Facet results are now typed as `IDictionary<string, IList<FacetResult>>` and hit highlights as `IDictionary<string, IList<string>>`. A quick way to resolve build errors introduced by this change is to add `using` aliases at the top of each file that uses the removed types. For example:

```
using FacetResults = System.Collections.Generic.IDictionary<string,
System.Collections.Generic.IList<Models.FacetResult>>;
using HitHighlights = System.Collections.Generic.IDictionary<string,
System.Collections.Generic.IList<string>>;
```

Change to SynonymMap

The `SynonymMap` constructor no longer has an `enum` parameter for `SynonymMapFormat`. This enum only had one value, and was therefore redundant. If you see build errors as a result of this, simply remove references to the `SynonymMapFormat` parameter.

Miscellaneous model class changes

The `AutocompleteMode` property of `AutocompleteParameters` is no longer nullable. If you have code that assigns this property to `null`, you can simply remove it and the property will automatically be initialized to the default value.

The order of the parameters to the `IndexAction` constructor has changed now that this constructor is auto-generated. Instead of using the constructor, we recommend using the factory methods `IndexAction.Upload`, `IndexAction.Merge`, and so on.

Removed preview features

If you are upgrading from version 8.0-preview to version 9, be aware that encryption with customer-managed keys has been removed since this feature is still in preview. Specifically, the `EncryptionKey` properties of `Index` and `SynonymMap` have been removed.

If your application has a hard dependency on this feature, you will not be able to upgrade to version 9 of the Azure Search .NET SDK. You can continue to use version 8.0-preview. However, please keep in mind that **we do not recommend using preview SDKs in production applications**. Preview features are for evaluation only and may change.

NOTE

If you created encrypted indexes or synonym maps using version 8.0-preview of the SDK, you will still be able use them and modify their definitions using version 9 of the SDK without adversely affecting their encryption status. Version 9 of the SDK will not send the `encryptionKey` property to the REST API, and as a result the REST API will not change the encryption status of the resource.

Behavioral change in data retrieval

If you're using the "dynamically typed" `Search`, `Suggest`, or `Get` APIs that return instances of type `Document`, be aware that they now deserialize empty JSON arrays to `object[]` instead of `string[]`.

Conclusion

If you need more details on using the Azure Search .NET SDK, see the [.NET How-to](#).

We welcome your feedback on the SDK. If you encounter problems, feel free to ask us for help on [Stack Overflow](#). If you find a bug, you can file an issue in the [Azure .NET SDK GitHub repository](#). Make sure to prefix your issue title with "[Azure Search]".

Thank you for using Azure Search!

Upgrade to Azure Search .NET SDK version 5

10/4/2020 • 5 minutes to read • [Edit Online](#)

If you're using version 4.0-preview or older of the [.NET SDK](#), this article will help you upgrade your application to use version 5.

For a more general walkthrough of the SDK including examples, see [How to use Azure Search from a .NET Application](#).

Version 5 of the Azure Search .NET SDK contains some changes from earlier versions. These are mostly minor, so changing your code should require only minimal effort. See [Steps to upgrade](#) for instructions on how to change your code to use the new SDK version.

NOTE

If you're using version 2.0-preview or older, you should upgrade to version 3 first, and then upgrade to version 5. See [Upgrading to the Azure Search .NET SDK version 3](#) for instructions.

Your Azure Search service instance supports several REST API versions, including the latest one. You can continue to use a version when it is no longer the latest one, but we recommend that you migrate your code to use the newest version. When using the REST API, you must specify the API version in every request via the `api-version` parameter. When using the .NET SDK, the version of the SDK you're using determines the corresponding version of the REST API. If you are using an older SDK, you can continue to run that code with no changes even if the service is upgraded to support a newer API version.

What's new in version 5

Version 5 of the Azure Search .NET SDK targets the latest generally available version of the Azure Search REST API, specifically 2017-11-11. This makes it possible to use new features of Azure Search from a .NET application, including the following:

- [Synonyms](#).
- You can now programmatically access warnings in indexer execution history (see the `Warning` property of `IndexerExecutionResult` in the [.NET reference](#) for more details).
- Support for .NET Core 2.
- New package structure supports using only the parts of the SDK that you need (see [Breaking changes in version 5](#) for details).

Steps to upgrade

First, update your NuGet reference for `Microsoft.Azure.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

Once NuGet has downloaded the new packages and their dependencies, rebuild your project. Depending on how your code is structured, it may rebuild successfully. If so, you're ready to go!

If your build fails, you should see a build error like the following:

```
The name 'SuggesterSearchMode' does not exist in the current context
```

The next step is to fix this build error. See [Breaking changes in version 5](#) for details on what causes the error and how to fix it.

Please note that due to changes in the packaging of the Azure Search .NET SDK, you must rebuild your application in order to use version 5. These changes are detailed in [Breaking changes in version 5](#).

You may see additional build warnings related to obsolete methods or properties. The warnings will include instructions on what to use instead of the deprecated feature. For example, if your application uses the `IndexingParametersExtensions.DoNotFailOnUnsupportedContentType` method, you should get a warning that says "This behavior is now enabled by default, so calling this method is no longer necessary."

Once you've fixed any build errors or warnings, you can make changes to your application to take advantage of new functionality if you wish. New features in the SDK are detailed in [What's new in version 5](#).

Breaking changes in version 5

New Package Structure

The most substantial breaking change in version 5 is that the `Microsoft.Azure.Search` assembly and its contents have been divided into four separate assemblies that are now distributed as four separate NuGet packages:

- `Microsoft.Azure.Search`: This is a meta-package that includes all the other Azure Search packages as dependencies. If you're upgrading from an earlier version of the SDK, simply upgrading this package and re-building should be enough to start using the new version.
- `Microsoft.Azure.Search.Data`: Use this package if you're developing a .NET application using Azure Search, and you only need to query or update documents in your indexes. If you also need to create or update indexes, synonym maps, or other service-level resources, use the `Microsoft.Azure.Search` package instead.
- `Microsoft.Azure.Search.Service`: Use this package if you're developing automation in .NET to manage Azure Search indexes, synonym maps, indexers, data sources, or other service-level resources. If you only need to query or update documents in your indexes, use the `Microsoft.Azure.Search.Data` package instead. If you need all the functionality of Azure Search, use the `Microsoft.Azure.Search` package instead.
- `Microsoft.Azure.Search.Common`: Common types needed by the Azure Search .NET libraries. You should not need to use this package directly in your application; It is only meant to be used as a dependency.

This change is technically breaking since many types were moved between assemblies. This is why rebuilding your application is necessary in order to upgrade to version 5 of the SDK.

There a small number of other breaking changes in version 5 that may require code changes in addition to rebuilding your application.

Change to Suggesters

The `Suggester` constructor no longer has an `enum` parameter for `SuggersterSearchMode`. This enum only had one value, and was therefore redundant. If you see build errors as a result of this, simply remove references to the `SuggersterSearchMode` parameter.

Removed obsolete members

You may see build errors related to methods or properties that were marked as obsolete in earlier versions and subsequently removed in version 5. If you encounter such errors, here is how to resolve them:

- If you were using the `IndexingParametersExtensions.IndexStorageMetadataOnly` method, use `SetBlobExtractionMode(BlobExtractionMode.StorageMetadata)` instead.
- If you were using the `IndexingParametersExtensions.SkipContent` method, use `SetBlobExtractionMode(BlobExtractionMode.AllMetadata)` instead.

Removed preview features

If you are upgrading from version 4.0-preview to version 5, be aware that JSON array and CSV parsing support for Blob Indexers has been removed since these features are still in preview. Specifically, the following methods of the `IndexingParametersExtensions` class have been removed:

- [ParseJsonArrays](#)
- [ParseDelimitedTextFiles](#)

If your application has a hard dependency on these features, you will not be able to upgrade to version 5 of the Azure Search .NET SDK. You can continue to use version 4.0-preview. However, please keep in mind that **we do not recommend using preview SDKs in production applications**. Preview features are for evaluation only and may change.

Conclusion

If you need more details on using the Azure Search .NET SDK, see the [.NET How-to](#).

We welcome your feedback on the SDK. If you encounter problems, feel free to ask us for help on [Stack Overflow](#). If you find a bug, you can file an issue in the [Azure .NET SDK GitHub repository](#). Make sure to prefix your issue title with "[Azure Search]".

Thank you for using Azure Search!

Upgrade to Azure Search .NET SDK version 3

10/4/2020 • 5 minutes to read • [Edit Online](#)

If you're using version 2.0-preview or older of the [Azure Search .NET SDK](#), this article will help you upgrade your application to use version 3.

For a more general walkthrough of the SDK including examples, see [How to use Azure Search from a .NET Application](#).

Version 3 of the Azure Search .NET SDK contains some changes from earlier versions. These are mostly minor, so changing your code should require only minimal effort. See [Steps to upgrade](#) for instructions on how to change your code to use the new SDK version.

NOTE

If you're using version 1.0.2-preview or older, you should upgrade to version 1.1 first, and then upgrade to version 3. See [Upgrading to the Azure Search .NET SDK version 1.1](#) for instructions.

Your Azure Search service instance supports several REST API versions, including the latest one. You can continue to use a version when it is no longer the latest one, but we recommend that you migrate your code to use the newest version. When using the REST API, you must specify the API version in every request via the `api-version` parameter. When using the .NET SDK, the version of the SDK you're using determines the corresponding version of the REST API. If you are using an older SDK, you can continue to run that code with no changes even if the service is upgraded to support a newer API version.

What's new in version 3

Version 3 of the Azure Search .NET SDK targets the latest generally available version of the Azure Search REST API, specifically 2016-09-01. This makes it possible to use many new features of Azure Search from a .NET application, including the following:

- [Custom analyzers](#)
- [Azure Blob Storage](#) and [Azure Table Storage](#) indexer support
- Indexer customization via [field mappings](#)
- ETags support to enable safe concurrent updating of index definitions, indexers, and data sources
- Support for building index field definitions declaratively by decorating your model class and using the new [FieldBuilder](#) class.
- Support for .NET Core and .NET Portable Profile 111

Steps to upgrade

First, update your NuGet reference for `Microsoft.Azure.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

Once NuGet has downloaded the new packages and their dependencies, rebuild your project. Depending on how your code is structured, it may rebuild successfully. If so, you're ready to go!

If your build fails, you should see a build error like the following:

```
Program.cs(31,45,31,86): error CS0266: Cannot implicitly convert type  
'Microsoft.Azure.Search.ISearchIndexClient' to 'Microsoft.Azure.Search.SearchIndexClient'. An explicit  
conversion exists (are you missing a cast?)
```

The next step is to fix this build error. See [Breaking changes in version 3](#) for details on what causes the error and how to fix it.

You may see additional build warnings related to obsolete methods or properties. The warnings will include instructions on what to use instead of the deprecated feature. For example, if your application uses the `IndexingParameters.Base64EncodeKeys` property, you should get a warning that says

`"This property is obsolete. Please create a field mapping using 'FieldMapping.Base64Encode' instead."`

Once you've fixed any build errors, you can make changes to your application to take advantage of new functionality if you wish. New features in the SDK are detailed in [What's new in version 3](#).

Breaking changes in version 3

There a small number of breaking changes in version 3 that may require code changes in addition to rebuilding your application.

Indexes.GetClient return type

The `Indexes.GetClient` method has a new return type. Previously, it returned `SearchIndexClient`, but this was changed to `ISearchIndexClient` in version 2.0-preview, and that change carries over to version 3. This is to support customers that wish to mock the `GetClient` method for unit tests by returning a mock implementation of `ISearchIndexClient`.

Example

If your code looks like this:

```
SearchIndexClient indexClient = serviceClient.Indexes.GetClient("hotels");
```

You can change it to this to fix any build errors:

```
ISearchIndexClient indexClient = serviceClient.Indexes.GetClient("hotels");
```

AnalyzerName, DataType, and others are no longer implicitly convertible to strings

There are many types in the Azure Search .NET SDK that derive from `ExtensibleEnum`. Previously these types were all implicitly convertible to type `string`. However, a bug was discovered in the `Object.Equals` implementation for these classes, and fixing the bug required disabling this implicit conversion. Explicit conversion to `string` is still allowed.

Example

If your code looks like this:

```

var customTokenizerName = TokenizerName.Create("my_tokenizer");
var customTokenFilterName = TokenFilterName.Create("my_tokenfilter");
var customCharFilterName = CharFilterName.Create("my_charfilter");

var index = new Index();
index.Analyzers = new Analyzer[]
{
    new CustomAnalyzer(
        "my_analyzer",
        customTokenizerName,
        new[] { customTokenFilterName },
        new[] { customCharFilterName }),
};

}

```

You can change it to this to fix any build errors:

```

const string CustomTokenizerName = "my_tokenizer";
const string CustomTokenFilterName = "my_tokenfilter";
const string CustomCharFilterName = "my_charfilter";

var index = new Index();
index.Analyzers = new Analyzer[]
{
    new CustomAnalyzer(
        "my_analyzer",
        CustomTokenizerName,
        new TokenFilterName[] { CustomTokenFilterName },
        new CharFilterName[] { CustomCharFilterName })
};

}

```

Removed obsolete members

You may see build errors related to methods or properties that were marked as obsolete in version 2.0-preview and subsequently removed in version 3. If you encounter such errors, here is how to resolve them:

- If you were using this constructor: `ScoringParameter(string name, string value)`, use this one instead:
`ScoringParameter(string name, IEnumerable<string> values)`
- If you were using the `ScoringParameter.Value` property, use the `ScoringParameter.Values` property or the `ToString` method instead.
- If you were using the `SearchRequestOptions.RequestId` property, use the `ClientRequestId` property instead.

Removed preview features

If you are upgrading from version 2.0-preview to version 3, be aware that JSON and CSV parsing support for Blob Indexers has been removed since these features are still in preview. Specifically, the following methods of the `IndexingParametersExtensions` class have been removed:

- `ParseJson`
- `ParseJsonArrays`
- `ParseDelimitedTextFiles`

If your application has a hard dependency on these features, you will not be able to upgrade to version 3 of the Azure Search .NET SDK. You can continue to use version 2.0-preview. However, please keep in mind that **we do not recommend using preview SDKs in production applications**. Preview features are for evaluation only and may change.

Conclusion

If you need more details on using the Azure Search .NET SDK, see the [.NET How-to](#).

We welcome your feedback on the SDK. If you encounter problems, feel free to ask us for help on [Stack Overflow](#). If you find a bug, you can file an issue in the [Azure .NET SDK GitHub repository](#). Make sure to prefix your issue title with "[Azure Search]".

Thank you for using Azure Search!

Upgrade to Azure Search .NET SDK version 1.1

10/4/2020 • 11 minutes to read • [Edit Online](#)

If you're using version 1.0.2-preview or older of the [Azure Search .NET SDK](#), this article will help you upgrade your application to use version 1.1.

For a more general walkthrough of the SDK including examples, see [How to use Azure Search from a .NET Application](#).

NOTE

Once you upgrade to version 1.1, or if you're already using a version between 1.1 and 2.0-preview inclusive, you should upgrade to version 3. See [Upgrading to the Azure Search .NET SDK version 3](#) for instructions.

First, update your NuGet reference for `Microsoft.Azure.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.

Once NuGet has downloaded the new packages and their dependencies, rebuild your project.

If you were previously using version 1.0.0-preview, 1.0.1-preview, or 1.0.2-preview, the build should succeed and you're ready to go!

If you were previously using version 0.13.0-preview or older, you should see build errors like the following:

```
Program.cs(137,56,137,62): error CS0117: 'Microsoft.Azure.Search.Models.IndexBatch' does not contain a definition for 'Create'  
Program.cs(137,99,137,105): error CS0117: 'Microsoft.Azure.Search.Models.IndexAction' does not contain a definition for 'Create'  
Program.cs(146,41,146,54): error CS1061: 'Microsoft.Azure.Search.IndexBatchException' does not contain a definition for 'IndexResponse' and no extension method 'IndexResponse' accepting a first argument of type 'Microsoft.Azure.Search.IndexBatchException' could be found (are you missing a using directive or an assembly reference?)  
Program.cs(163,13,163,42): error CS0246: The type or namespace name 'DocumentSearchResponse' could not be found (are you missing a using directive or an assembly reference?)
```

The next step is to fix the build errors one by one. Most will require changing some class and method names that have been renamed in the SDK. [List of breaking changes in version 1.1](#) contains a list of these name changes.

If you're using custom classes to model your documents, and those classes have properties of non-nullable primitive types (for example, `int` or `bool` in C#), there is a bug fix in the 1.1 version of the SDK of which you should be aware. See [Bug fixes in version 1.1](#) for more details.

Finally, once you've fixed any build errors, you can make changes to your application to take advantage of new functionality if you wish.

List of breaking changes in version 1.1

The following list is ordered by the likelihood that the change will affect your application code.

IndexBatch and IndexAction changes

`IndexBatch.Create` has been renamed to `IndexBatch.New` and no longer has a `params` argument. You can use `IndexBatch.New` for batches that mix different types of actions (merges, deletes, etc.). In addition, there are new static methods for creating batches where all the actions are the same: `Delete`, `Merge`, `MergeOrUpload`, and

`Upload`.

`IndexAction` no longer has public constructors and its properties are now immutable. You should use the new static methods for creating actions for different purposes: `Delete`, `Merge`, `MergeOrUpload`, and `Upload`. `IndexAction.Create` has been removed. If you used the overload that takes only a document, make sure to use `Upload` instead.

Example

If your code looks like this:

```
var batch = IndexBatch.Create(documents.Select(doc => IndexAction.Create(doc)));
indexClient.Documents.Index(batch);
```

You can change it to this to fix any build errors:

```
var batch = IndexBatch.New(documents.Select(doc => IndexAction.Upload(doc)));
indexClient.Documents.Index(batch);
```

If you want, you can further simplify it to this:

```
var batch = IndexBatch.Upload(documents);
indexClient.Documents.Index(batch);
```

IndexBatchException changes

The `IndexBatchException.IndexResponse` property has been renamed to `IndexingResults`, and its type is now `IList<IndexingResult>`.

Example

If your code looks like this:

```
catch (IndexBatchException e)
{
    Console.WriteLine(
        "Failed to index some of the documents: {0}",
        String.Join(", ", e.IndexResponse.Results.Where(r => !r.Succeeded).Select(r => r.Key)));
}
```

You can change it to this to fix any build errors:

```
catch (IndexBatchException e)
{
    Console.WriteLine(
        "Failed to index some of the documents: {0}",
        String.Join(", ", e.IndexingResults.Where(r => !r.Succeeded).Select(r => r.Key)));
}
```

Operation method changes

Each operation in the Azure Search .NET SDK is exposed as a set of method overloads for synchronous and asynchronous callers. The signatures and factoring of these method overloads has changed in version 1.1.

For example, the "Get Index Statistics" operation in older versions of the SDK exposed these signatures:

In `IIndexOperations`:

```
// Asynchronous operation with all parameters
Task<IndexGetStatisticsResponse> GetStatisticsAsync(
    string indexName,
    CancellationToken cancellationToken);
```

In `IndexOperationsExtensions` :

```
// Asynchronous operation with only required parameters
public static Task<IndexGetStatisticsResponse> GetStatisticsAsync(
    this IIIndexOperations operations,
    string indexName);

// Synchronous operation with only required parameters
public static IndexGetStatisticsResponse GetStatistics(
    this IIIndexOperations operations,
    string indexName);
```

The method signatures for the same operation in version 1.1 look like this:

In `IIndexesOperations` :

```
// Asynchronous operation with lower-level HTTP features exposed
Task<AzureOperationResponse<IndexGetStatisticsResult>> GetStatisticsWithHttpMessagesAsync(
    string indexName,
    SearchRequestOptions searchRequestOptions = default(SearchRequestOptions),
    Dictionary<string, List<string>> customHeaders = null,
    CancellationToken cancellationToken = default(CancellationToken));
```

In `IndexesOperationsExtensions` :

```
// Simplified asynchronous operation
public static Task<IndexGetStatisticsResult> GetStatisticsAsync(
    this IIndexesOperations operations,
    string indexName,
    SearchRequestOptions searchRequestOptions = default(SearchRequestOptions),
    CancellationToken cancellationToken = default(CancellationToken));

// Simplified synchronous operation
public static IndexGetStatisticsResult GetStatistics(
    this IIndexesOperations operations,
    string indexName,
    SearchRequestOptions searchRequestOptions = default(SearchRequestOptions));
```

Starting with version 1.1, the Azure Search .NET SDK organizes operation methods differently:

- Optional parameters are now modeled as default parameters rather than additional method overloads. This reduces the number of method overloads, sometimes dramatically.
- The extension methods now hide a lot of the extraneous details of HTTP from the caller. For example, older versions of the SDK returned a response object with an HTTP status code, which you often didn't need to check because operation methods throw `CloudException` for any status code that indicates an error. The new extension methods just return model objects, saving you the trouble of having to unwrap them in your code.
- Conversely, the core interfaces now expose methods that give you more control at the HTTP level if you need it. You can now pass in custom HTTP headers to be included in requests, and the new `AzureOperationResponse<T>` return type gives you direct access to the `HttpRequestMessage` and `HttpResponseMessage` for the operation. `AzureOperationResponse` is defined in the `Microsoft.Rest.Azure` namespace and replaces `Hyak.Common.OperationResponse`.

ScoringParameters changes

A new class named `ScoringParameter` has been added in the latest SDK to make it easier to provide parameters to scoring profiles in a search query. Previously the `ScoringProfiles` property of the `SearchParameters` class was typed as `IList<string>`; Now it is typed as `IList<ScoringParameter>`.

Example

If your code looks like this:

```
var sp = new SearchParameters();
sp.ScoringProfile = "jobsScoringFeatured";           // Use a scoring profile
sp.ScoringParameters = new[] { "featuredParam-featured", "mapCenterParam-" + lon + "," + lat };
```

You can change it to this to fix any build errors:

```
var sp = new SearchParameters();
sp.ScoringProfile = "jobsScoringFeatured";           // Use a scoring profile
sp.ScoringParameters =
{
    new ScoringParameter("featuredParam", new[] { "featured" }),
    new ScoringParameter("mapCenterParam", GeographyPoint.Create(lat, lon))
};
```

Model class changes

Due to the signature changes described in [Operation method changes](#), many classes in the `Microsoft.Azure.Search.Models` namespace have been renamed or removed. For example:

- `IndexDefinitionResponse` has been replaced by `AzureOperationResponse<Index>`
- `DocumentSearchResponse` has been renamed to `DocumentSearchResult`
- `IndexResult` has been renamed to `IndexingResult`
- `Documents.Count()` now returns a `long` with the document count instead of a `DocumentCountResponse`
- `IndexGetStatisticsResponse` has been renamed to `IndexGetStatisticsResult`
- `IndexListResponse` has been renamed to `IndexListResult`

To summarize, `OperationResponse`-derived classes that existed only to wrap a model object have been removed. The remaining classes have had their suffix changed from `Response` to `Result`.

Example

If your code looks like this:

```
IndexerGetStatusResponse statusResponse = null;

try
{
    statusResponse = _searchClient.Indexers.GetStatus(indexer.Name);
}
catch (Exception ex)
{
    Console.WriteLine("Error polling for indexer status: {0}", ex.Message);
    return;
}

IndexerExecutionResult lastResult = statusResponse.ExecutionInfo.LastResult;
```

You can change it to this to fix any build errors:

```

IndexerExecutionInfo status = null;

try
{
    status = _searchClient.Indexers.GetStatus(indexer.Name);
}
catch (Exception ex)
{
    Console.WriteLine("Error polling for indexer status: {0}", ex.Message);
    return;
}

IndexerExecutionResult lastResult = status.LastResult;

```

Response classes and `IEnumerable`

An additional change that may affect your code is that response classes that hold collections no longer implement `IEnumerable<T>`. Instead, you can access the collection property directly. For example, if your code looks like this:

```

DocumentSearchResponse<Hotel> response = indexClient.Documents.Search<Hotel>(searchText, sp);
foreach (SearchResult<Hotel> result in response)
{
    Console.WriteLine(result.Document);
}

```

You can change it to this to fix any build errors:

```

DocumentSearchResult<Hotel> response = indexClient.Documents.Search<Hotel>(searchText, sp);
foreach (SearchResult<Hotel> result in response.Results)
{
    Console.WriteLine(result.Document);
}

```

Special case for web applications

If you have a web application that serializes `DocumentSearchResponse` directly to send search results to the browser, you will need to change your code or the results will not serialize correctly. For example, if your code looks like this:

```

public ActionResult Search(string q = "")
{
    // If blank search, assume they want to search everything
    if (string.IsNullOrWhiteSpace(q))
        q = "*";

    return new JsonResult
    {
        JsonRequestBehavior = JsonRequestBehavior.AllowGet,
        Data = _featuresSearch.Search(q)
    };
}

```

You can change it by getting the `.Results` property of the search response to fix search result rendering:

```

public ActionResult Search(string q = "")
{
    // If blank search, assume they want to search everything
    if (string.IsNullOrWhiteSpace(q))
        q = "*";

    return new JsonResult
    {
        JsonRequestBehavior = JsonRequestBehavior.AllowGet,
        Data = _featuresSearch.Search(q).Results
    };
}

```

You will have to look for such cases in your code yourself; **The compiler will not warn you** because `JsonResult.Data` is of type `object`.

CloudException changes

The `CloudException` class has moved from the `Hyak.Common` namespace to the `Microsoft.Rest.Azure` namespace. Also, its `Error` property has been renamed to `Body`.

SearchServiceClient and SearchIndexClient changes

The type of the `Credentials` property has changed from `SearchCredentials` to its base class, `ServiceClientCredentials`. If you need to access the `SearchCredentials` of a `SearchIndexClient` or `SearchServiceClient`, please use the new `SearchCredentials` property.

In older versions of the SDK, `SearchServiceClient` and `SearchIndexClient` had constructors that took an `HttpClient` parameter. These have been replaced with constructors that take an `HttpClientHandler` and an array of `DelegatingHandler` objects. This makes it easier to install custom handlers to pre-process HTTP requests if necessary.

Finally, the constructors that took a `Uri` and `SearchCredentials` have changed. For example, if you have code that looks like this:

```

var client =
    new SearchServiceClient(
        new SearchCredentials("abc123"),
        new Uri("http://myservice.search.windows.net"));

```

You can change it to this to fix any build errors:

```

var client =
    new SearchServiceClient(
        new Uri("http://myservice.search.windows.net"),
        new SearchCredentials("abc123"));

```

Also note that the type of the `credentials` parameter has changed to `ServiceClientCredentials`. This is unlikely to affect your code since `SearchCredentials` is derived from `ServiceClientCredentials`.

Passing a request ID

In older versions of the SDK, you could set a request ID on the `SearchServiceClient` or `SearchIndexClient` and it would be included in every request to the REST API. This is useful for troubleshooting issues with your search service if you need to contact support. However, it is more useful to set a unique request ID for every operation rather than to use the same ID for all operations. For this reason, the `SetClientRequestId` methods of `SearchServiceClient` and `SearchIndexClient` have been removed. Instead, you can pass a request ID to each operation method via the optional `SearchRequestOptions` parameter.

NOTE

In a future release of the SDK, we will add a new mechanism for setting a request ID globally on the client objects that is consistent with the approach used by other Azure SDKs.

Example

If you have code that looks like this:

```
client.SetClientRequestId(Guid.NewGuid());  
...  
long count = client.Documents.Count();
```

You can change it to this to fix any build errors:

```
long count = client.Documents.Count(new SearchRequestOptions(requestId: Guid.NewGuid()));
```

Interface name changes

The operation group interface names have all changed to be consistent with their corresponding property names:

- The type of `ISearchServiceClient.Indexes` has been renamed from `IIndexOperations` to `IIndexesOperations`.
- The type of `ISearchServiceClient.Indexers` has been renamed from `IIndexerOperations` to `IIndexersOperations`.
- The type of `ISearchServiceClient.DataSources` has been renamed from `IDataSourceOperations` to `IDataSourcesOperations`.
- The type of `ISearchIndexClient.Documents` has been renamed from `IDocumentOperations` to `IDocumentsOperations`.

This change is unlikely to affect your code unless you created mocks of these interfaces for test purposes.

Bug fixes in version 1.1

There was a bug in older versions of the Azure Search .NET SDK relating to serialization of custom model classes. The bug could occur if you created a custom model class with a property of a non-nullable value type.

Steps to reproduce

Create a custom model class with a property of non-nullable value type. For example, add a public `UnitCount` property of type `int` instead of `int?`.

If you index a document with the default value of that type (for example, 0 for `int`), the field will be null in Azure Search. If you subsequently search for that document, the `Search` call will throw `JsonSerializationException` complaining that it can't convert `null` to `int`.

Also, filters may not work as expected since null was written to the index instead of the intended value.

Fix details

We have fixed this issue in version 1.1 of the SDK. Now, if you have a model class like this:

```
public class Model
{
    public string Key { get; set; }

    public int IntValue { get; set; }
}
```

and you set `IntValue` to 0, that value is now correctly serialized as 0 on the wire and stored as 0 in the index. Round tripping also works as expected.

There is one potential issue to be aware of with this approach: If you use a model type with a non-nullable property, you have to **guarantee** that no documents in your index contain a null value for the corresponding field. Neither the SDK nor the Azure Search REST API will help you to enforce this.

This is not just a hypothetical concern: Imagine a scenario where you add a new field to an existing index that is of type `Edm.Int32`. After updating the index definition, all documents will have a null value for that new field (since all types are nullable in Azure Search). If you then use a model class with a non-nullable `int` property for that field, you will get a `JsonSerializationException` like this when trying to retrieve documents:

```
Error converting value {null} to type 'System.Int32'. Path 'IntValue'.
```

For this reason, we still recommend that you use nullable types in your model classes as a best practice.

For more details on this bug and the fix, please see [this issue on GitHub](#).

Upgrading versions of the Azure Search .NET Management SDK

10/4/2020 • 3 minutes to read • [Edit Online](#)

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

SDK VERSION	CORRESPONDING REST API VERSION	FEATURE ADDITION OR BEHAVIOR CHANGE
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. Update your NuGet reference for `Microsoft.Azure.Management.Search` using either the NuGet Package Manager Console or by right-clicking on your project references and selecting "Manage NuGet Packages..." in Visual Studio.
2. Once NuGet has downloaded the new packages and their dependencies, rebuild your project. Depending on how your code is structured, it may rebuild successfully, in which case you are done.
3. If your build fails, it could be because you've implemented some of the SDK interfaces (for example, for the purposes of unit testing), which have changed. To resolve this, you'll need to implement newer methods such as `BeginCreateOrUpdateWithHttpMessagesAsync`.
4. After fixing any build errors, you can make changes to your application to take advantage of new functionality.

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 should not be visible on the public internet.

New APIs

API	CATEGORY	DETAILS

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 (Azure Storage, 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 do not 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

10/4/2020 • 7 minutes to read • [Edit Online](#)

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 are 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 does not 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 are no breaking changes, but there are a few 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 is equivalent to the generally available version. The only code change required is modifying the api-version.

Behavior changes include the following:

- [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 leverages 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 are using a [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 was not 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](#). You should be able to replace the skill reference with no other changes. The API signature is the same for both versions.

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 sub-fields 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 this applies to you, you will need to redesign your schema to fit within the new limits and then rebuild your index.
- There is 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 this applies to you, you will 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:

```
{  
    "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 sub-fields 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 since there is no sample code available at the time of this writing.
3. Perform a PUT request to update the index to the new format. Make sure not to change any other details of the index such as the searchability/filterability of fields, since this is not 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](#)

Configure customer-managed keys for data encryption in Azure Cognitive Search

10/4/2020 • 11 minutes to read • [Edit Online](#)

Azure Cognitive Search automatically encrypts indexed content at rest with [service-managed keys](#). If more protection is needed, you can supplement default encryption with an additional encryption layer using keys that you create and manage in Azure Key Vault. This article walks you through the steps of setting up CMK encryption.

CMK encryption is dependent 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's APIs to generate encryption keys. With Azure Key Vault, you can also audit key usage if you [enable logging](#).

Encryption with customer-managed keys is applied to individual indexes or synonym maps when those objects are created, and is not specified on the search service level itself. Only new objects can be encrypted. You cannot encrypt content that already exists.

Keys don't all need to be in the same key vault. A single search service can host multiple encrypted indexes or synonym maps, each encrypted with their own customer-managed encryption keys, stored in different key vaults. You can also have indexes and synonym maps in the same service that are not encrypted using customer-managed keys.

Double encryption

For services created after August 1, 2020 and in specific regions, the scope of CMK encryption includes temporary disks, achieving [full double encryption](#), currently available in these regions:

- West US 2
- East US
- South Central US
- US Gov Virginia
- US Gov Arizona

If you are using a different region, or a service created prior to August 1, then your CMK encryption is limited to just the data disk, excluding the temporary disks used by the service.

Prerequisites

The following services and services are used in this example.

- [Create an Azure Cognitive Search service](#) or [find an existing service](#).
- [Create an Azure Key Vault resource](#) or find an existing vault in the same subscription as Azure Cognitive Search. This feature has a same-subscription requirement.
- [Azure PowerShell](#) or [Azure CLI](#) is used for configuration tasks.
- [Postman](#), [Azure PowerShell](#) and [.NET SDK preview](#) can be used to call the REST API that creates indexes and synonym maps that include an encryption key parameter. There is no portal support for adding a key to indexes or synonym maps at this time.

NOTE

Due to the nature of encryption with customer-managed keys, Azure Cognitive Search will not be able to 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 will only encounter issues if you purposely disabled it. Purge protection is not enabled by default, but it is required for Azure Cognitive Search CMK encryption. For more information, see [soft-delete](#) and [purge protection](#) overviews.

1 - Enable key recovery

The key vault must have **soft-delete** and **purge protection** enabled. You can set those features using the portal or the following PowerShell or Azure CLI commands.

Using PowerShell

1. Run `Connect-AzAccount` to set up your Azure credentials.
2. Run the following command to connect to your key vault, replacing `<vault_name>` with a valid name:

```
$resource = Get-AzResource -ResourceId (Get-AzKeyVault -VaultName "<vault_name>").ResourceId
```

3. Azure Key Vault is created with soft-delete enabled. If it's disabled on your vault, run the following command:

```
$resource.Properties | Add-Member -MemberType NoteProperty -Name "enableSoftDelete" -Value 'true'
```

4. Enable purge protection:

```
$resource.Properties | Add-Member -MemberType NoteProperty -Name "enablePurgeProtection" -Value 'true'
```

5. Save your updates:

```
Set-AzResource -resourceid $resource.ResourceId -Properties $resource.Properties
```

Using Azure CLI

```
az keyvault update -n <vault_name> -g <resource_group> --enable-soft-delete --enable-purge-protection
```

2 - Create a new key

If you are using an existing key to encrypt Azure Cognitive Search content, skip this step.

1. [Sign in to Azure portal](#) and open your key vault overview page.
2. Select the **Keys** setting from the left navigation pane, and click **+ 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. Click on the **Create** button to start the deployment.

Make a note of the Key Identifier – this is composed from the **key value Uri**, the **key name**, and the **key version**. You will need these to define an encrypted index in Azure Cognitive Search.

The screenshot shows the 'Properties' page for a key in Azure Key Vault. At the top, there's a key icon and the identifier '208d83db2625450cba22f5bd3ec38027'. Below that is a 'Key Version' section with 'Save' and 'Discard' buttons. The main area contains 'Properties' and 'Key Identifier' sections. The 'Key Identifier' section displays the URL 'https://asvault01.vault.local.azurestack.external/keys/ASKey01/208d83db2625450cba22f5bd3ec38027', which is highlighted with a red box. Below this are 'Settings' (activation and expiration date checkboxes), an 'Enabled' switch set to 'Yes', and a 'Tags' section with '0 tags'. Under 'Permitted operations', several checkboxes are checked: Encrypt, Sign, Wrap Key, Decrypt, Verify, and Unwrap Key. The 'Sign' and 'Wrap Key' checkboxes are also highlighted with a red box.

3 - Create a service identity

Assigning an identity to your search service enables you to grant Key Vault access permissions to your search service. Your search service will use its identity to authenticate with Azure Key vault.

Azure Cognitive Search supports two ways for assigning identity: a managed identity or an externally-managed Azure Active Directory application.

If possible, use a managed identity. It is the simplest way of assigning an identity to your search service and should work in most scenarios. If you are using multiple keys for indexes and synonym maps, or if your solution is in a distributed architecture that disqualifies identity-based authentication, use the advanced [externally-managed Azure Active Directory approach](#) described at the end of this article.

In general, a managed identity enables your search service to authenticate to Azure Key Vault without storing credentials 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. [Learn more about Managed identities](#).

1. [Sign in to Azure portal](#) and open your search service overview page.
2. Click **Identity** in the left navigation pane, change its status to **On**, and click **Save**.

The screenshot shows the Azure portal interface. On the left, a navigation pane lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Keys, Scale, Search traffic analytics, and Identity. The 'Identity' option is selected and highlighted with a red box. The main content area is titled 'System assigned' and contains a note about managed identities. It features three buttons at the top: Save (highlighted with a red box), Discard, and Refresh. Below that is a 'Status' section with two options: 'Off' and 'On' (highlighted with a red box). At the bottom are Save, Discard, and Refresh buttons.

4 - Grant key access permissions

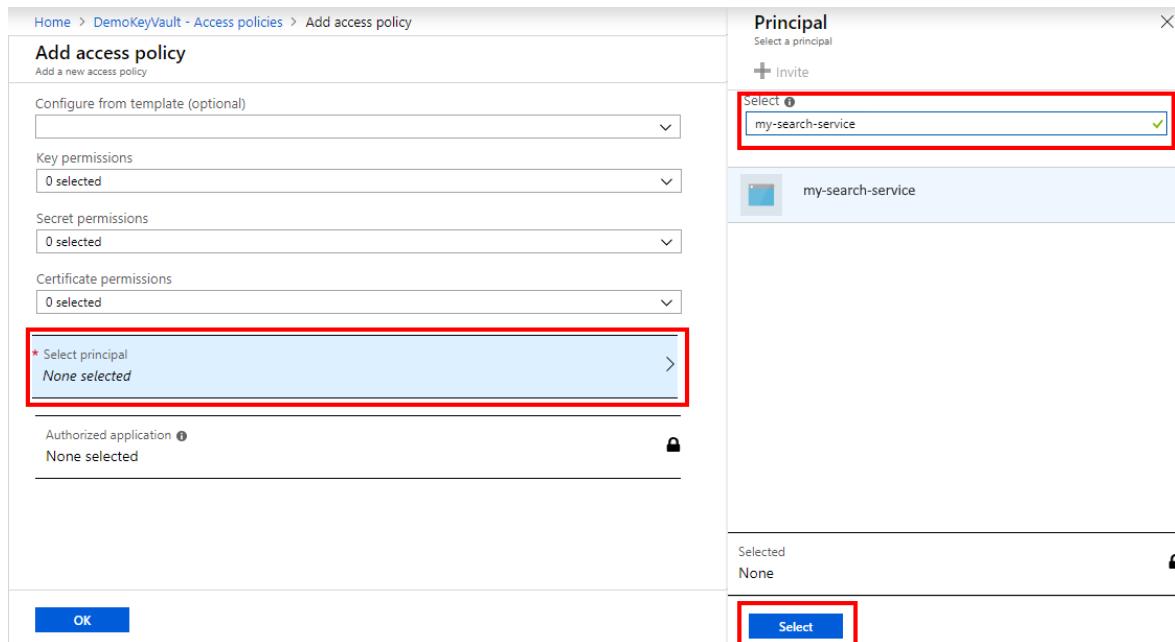
To enable your search service to use your Key Vault key, you'll need to grant your search service certain access permissions.

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. [Sign in to Azure portal](#) and open your key vault overview page.
2. Select the **Access policies** setting from the left navigation pane, and click **+ Add new**.

The screenshot shows the 'DemoKeyVault - Access policies' page. The left navigation pane includes Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Keys, Secrets, Certificates), and Access policies (which is selected and highlighted with a red box). The main content area has a button labeled '+ Add new' (highlighted with a red box) and a link to 'Click to show advanced access policies'. At the top are Save, Discard, and Refresh buttons.

3. Click **Select principal** and select your Azure Cognitive Search service. You can search for it by name or by the object ID that was displayed after enabling managed identity.



4. Click on **Key permissions** and select *Get*, *Unwrap Key* and *Wrap Key*. You can use the *Azure Data Lake Storage* or *Azure Storage* template to quickly select the required permissions.

Azure Cognitive Search must be granted with the following [access permissions](#):

- *Get* - allows your search service to retrieve the public parts of your key in a Key Vault
- *Wrap Key* - allows your search service to use your key to protect the internal encryption key
- *Unwrap Key* - allows your search service to use your key to unwrap the internal encryption key

Add access policy

Add a new access policy

Configure from template (optional)

Azure Data Lake Storage or Azure Storage

★ Select principal
my-search-service

Key permissions

3 selected

Select all

Key Management Operations

Get

List

Update

Create

Import

Delete

Recover

Backup

Restore

Cryptographic Operations

Decrypt

Encrypt

Unwrap Key

Wrap Key

Verify

Sign

Privileged Key Operations

Purge

OK

5. For **Secret Permissions**, select *Get*.
6. For **Certificate Permissions**, select *Get*.
7. Click **OK** and **Save** the access policy changes.

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 the new key\version before deleting the previous key\version. Failing to do so will render the index or synonym map unusable, at you won't be able to decrypt the content once key access is lost.

5 - Encrypt content

To add a customer-managed key on an index or synonym map, you must use the [Search REST API](#) or an SDK. The portal does not expose synonym maps or encryption properties. When you use a valid API, both indexes and synonym maps support a top-level **encryptionKey** property..

Using the **key vault Uri**, **key name** and the **key version** of your Key vault key, create an **encryptionKey** definition as follows:

```
{  
  "encryptionKey": {  
    "keyVaultUri": "https://demokeyvault.vault.azure.net",  
    "keyVaultKeyName": "myEncryptionKey",  
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660"  
  }  
}
```

NOTE

None of these key vault details are considered secret and could be easily retrieved by browsing to the relevant Azure Key Vault key page in Azure portal.

If you are using an AAD application for Key Vault authentication instead of using a managed identity, add the AAD application **access credentials** to your encryption key:

```
{  
  "encryptionKey": {  
    "keyVaultUri": "https://demokeyvault.vault.azure.net",  
    "keyVaultKeyName": "myEncryptionKey",  
    "keyVaultKeyVersion": "eaab6a663d59439ebb95ce2fe7d5f660",  
    "accessCredentials": {  
      "applicationId": "00000000-0000-0000-000000000000",  
      "applicationSecret": "myApplicationSecret"  
    }  
  }  
}
```

Example: Index encryption

The details of creating a new index via the REST API could be found at [Create Index \(Azure Cognitive Search REST API\)](#), where the only difference here is specifying the encryption key details as part of the index definition:

```
{
  "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"
  }
}
```

You can now send the index creation request, and then start using the index normally.

Example: Synonym map encryption

The details of creating a new synonym map via the REST API can be found at [Create Synonym Map \(Azure Cognitive Search REST API\)](#), where the only difference here is specifying the encryption key details as part of the synonym map definition:

```
{
  "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"
  }
}
```

You can now send the synonym map creation request, and then start using it normally.

IMPORTANT

While **encryptionKey** cannot be added to existing Azure Cognitive 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 Azure Cognitive 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.

Restoring Key vault access permissions at a later time will restore content access.

Advanced: Use an externally managed Azure Active Directory application

When a managed identity is not possible, you can create an Azure Active Directory application with a security principal for your Azure Cognitive Search service. Specifically, a managed identity is not viable under these conditions:

- You cannot 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\synonym maps, each using a different key from a different Key vault, where each key vault must use a **different identity** for authentication. If using a different identity to manage different Key vaults is not a requirement, consider using the managed identity option above.

To accommodate such topologies, Azure Cognitive Search supports using Azure Active Directory (AAD) applications for authentication between your search service and Key Vault.

To create an AAD application in the portal:

1. [Create an Azure Active Directory application](#).
2. [Get the application ID and authentication key](#) as those will be required for creating an encrypted index.
Values you will need to provide include **application ID** and **authentication key**.

IMPORTANT

When deciding to use an AAD application of authentication instead of a managed identity, consider the fact that Azure Cognitive Search is not authorized to manage your AAD application on your behalf, and it is up to you to manage your AAD application, such as periodic rotation of the application authentication key. When changing an AAD application or its authentication key, any Azure Cognitive Search index or synonym map that uses that application must first be updated to use the new application ID\key before deleting the previous application or its authorization key, and before revoking your Key Vault access to it. 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.

Work with encrypted content

With CMK encryption, you will notice latency for both indexing and queries due to the extra encrypt/decrypt work. Azure Cognitive Search does not log encryption activity, but you can monitor key access through key vault logging. We recommend that you [enable logging](#) as part of key vault set up.

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 cannot decrypt content, you will get this message: "Access forbidden. The query key used might have been revoked - please retry."

Next steps

If you are unfamiliar with Azure security architecture, review the [Azure Security documentation](#), and in particular, this article:

[Data encryption-at-rest](#)

Get customer-managed key information from indexes and synonym maps

10/4/2020 • 2 minutes to read • [Edit Online](#)

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, use the REST API or an SDK to retrieve the **encryptionKey** property from an index or synonym map definition.

We recommend that you [enable logging](#) on Key Vault so that you can monitor key usage.

Get the admin API key

To get object definitions from a search service, you will need to authenticate with admin rights. 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, click **Keys** and copy an admin API. An admin key is required for index and synonym map retrieval.

For the remaining steps, switch to PowerShell and the REST API. The portal does not show synonym maps, nor the encryption key properties of indexes.

Use PowerShell and REST

Run the following commands to set up the variables and get object definitions.

```
<# Connect to Azure #>
$Connect-AzAccount

<# Provide the admin API key used for search service authentication  #>
$headers = @{
    'api-key' = '<YOUR-ADMIN-API-KEY>'
    'Content-Type' = 'application/json'
    'Accept' = 'application/json' }

<# List all existing synonym maps #>
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms?api-version=2020-06-30&$select=name'
Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json

<# List all existing indexes #>
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes?api-version=2020-06-30&$select=name'
Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json

<# Return a specific synonym map definition. The encryptionKey property is at the end #>
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/synonyms/<YOUR-SYNONYM-MAP-NAME>?api-version=2020-06-30'
Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json

<# Return a specific index definition. The encryptionKey property is at the end #>
$url = 'https://<YOUR-SEARCH-SERVICE>.search.windows.net/indexes/<YOUR-INDEX-NAME>?api-version=2020-06-30'
Invoke-RestMethod -Uri $url -Headers $headers | ConvertTo-Json
```

Next steps

Now that you know which encryption key and version is used, you can manage the key in Azure Key Vault or check other configuration settings.

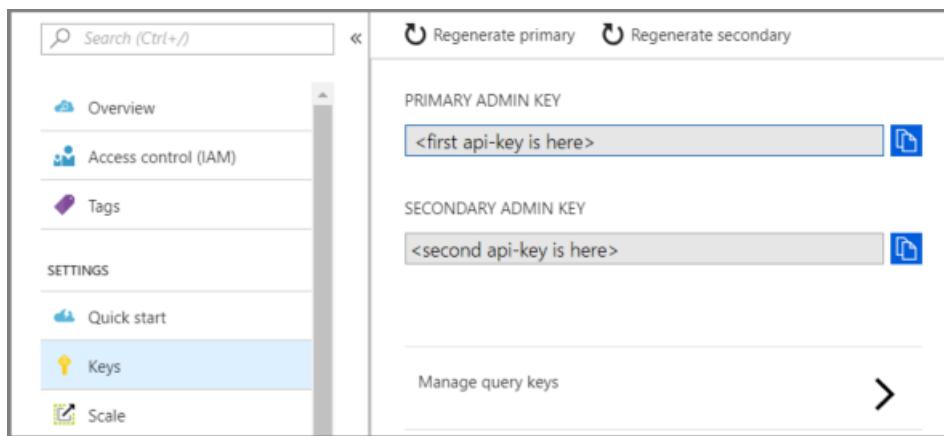
- [Quickstart: Set and retrieve a secret from Azure Key Vault using PowerShell](#)
- [Configure customer-managed keys for data encryption in Azure Cognitive Search](#)

Create and manage api-keys for an Azure Cognitive Search service

10/4/2020 • 5 minutes to read • [Edit Online](#)

All requests to a search service need a read-only api-key that was generated specifically for your service. The api-key is the sole mechanism for authenticating access to your search service endpoint and must be included on every request. In [REST solutions](#), the api-key is typically specified in a request header. In [.NET solutions](#), a key is often specified as a configuration setting and then passed as [Credentials](#) (admin key) or [SearchCredentials](#) (query key) on [SearchServiceClient](#).

Keys are created with your search service during service provisioning. You can view and obtain key values in the [Azure portal](#).



What is an api-key

An api-key is a string composed of randomly generated numbers and letters. Through [role-based permissions](#), you can delete or read the keys, but you can't replace a key with a user-defined password or use Active Directory as the primary authentication methodology for accessing search operations.

Two types of keys are used to access your search service: admin (read-write) and query (read-only).

KEY	DESCRIPTION	LIMITS
Admin	<p>Grants full rights to all operations, including the ability to manage the service, create and delete indexes, indexers, and data sources.</p> <p>Two admin keys, referred to as <i>primary</i> and <i>secondary</i> keys in the portal, are generated when the service is created and can be individually regenerated on demand. Having two keys allows you to roll over one key while using the second key for continued access to the service.</p> <p>Admin keys are only specified in HTTP request headers. You cannot place an admin api-key in a URL.</p>	Maximum of 2 per service

KEY	DESCRIPTION	LIMITS
Query	<p>Grants read-only access to indexes and documents, and are typically distributed to client applications that issue search requests.</p> <p>Query keys are created on demand. You can create them manually in the portal or programmatically via the Management REST API.</p> <p>Query keys can be specified in an HTTP request header for search, suggestion, or lookup operation. Alternatively, you can pass a query key as a parameter on a URL. Depending on how your client application formulates the request, it might be easier to pass the key as a query parameter:</p> <div style="border: 1px solid #ccc; padding: 5px;"><pre>GET /indexes/hotels/docs? search=*&\$orderby=lastRenovationDate desc&api-version=2020-06-30&api-key= [query key]</pre></div>	50 per service

Visually, there is no distinction between an admin key or query key. Both keys are strings composed of 32 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](#) or use the [REST API](#) to return the value and key type.

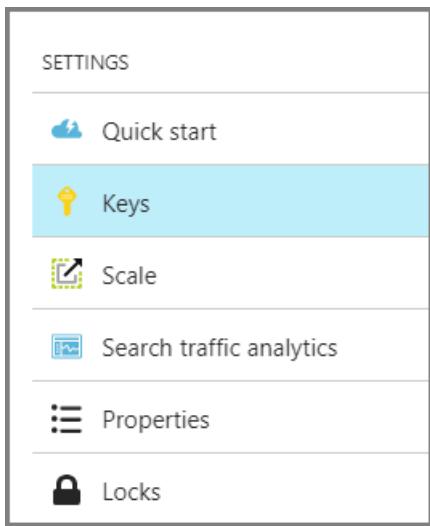
NOTE

It is considered a poor security practice to pass sensitive data such as an `api-key` in the request URL. For this reason, Azure Cognitive Search only accepts a query key as an `api-key` in the query string, and you should avoid doing so unless the contents of your index should be publicly available. As a general rule, we recommend passing your `api-key` as a request header.

Find existing keys

You can obtain access keys in the portal or through the [Management REST API](#). For more information, see [Manage admin and query api-keys](#).

1. Sign in to the [Azure portal](#).
 2. List the [search services](#) for your subscription.
 3. Select the service and on the Overview page, click **Settings > 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.

1. Sign in to the [Azure portal](#).
2. List the [search services](#) for your subscription.
3. Select the service and on the Overview page, click **Settings > Keys**.
4. Click **Manage query keys**.
5. Use the query key already generated for your service, or create up to 50 new query keys. The default query key is not named, but additional query keys can be named for manageability.

A screenshot of the 'Manage query keys' interface in the Azure portal. The top navigation bar shows 'Dashboard > mydemo - Keys > Manage query keys'. The main title is 'Manage query keys' with a 'mydemo' service name. Below it is a button labeled '+ Add'. To the right of the button is the text 'Use existing or create new'. Below the button is a table with two columns: 'NAME' and 'KEY'. The 'NAME' column contains the placeholder '<empty>'. The 'KEY' column contains the placeholder '<auto-generated-alphanumeric-string>'. Both the '+ Add' button and the 'KEY' column are highlighted with red boxes.

NOTE

A code example showing query key usage can be found in [Query an Azure Cognitive Search index in C#](#).

Regenerate admin keys

Two admin keys are created for each service so that you can rotate a primary key, using the secondary key for business continuity.

1. In the **Settings >Keys** page, 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 is not deleted and you are not locked out permanently.

You can still access the service through the portal or the management layer ([REST API](#), [PowerShell](#), or Azure Resource Manager). Management functions are operative through a subscription ID not a service api-key, and thus still available even if your api-keys are not.

After you create new keys via portal or management layer, access is restored to your content (indexes, indexers, data sources, synonym maps) once you have the new keys and provide those keys on requests.

Secure api-keys

Key security is ensured by restricting access via the portal or Resource Manager interfaces (PowerShell or command-line interface). As noted, subscription administrators can view and regenerate all api-keys. As a precaution, review role assignments to understand who has access to the admin keys.

- In the service dashboard, click **Access control (IAM)** and then the **Role assignments** tab to view role assignments for your service.

Members of the following roles can view and regenerate keys: Owner, Contributor, [Search Service Contributors](#)

NOTE

For identity-based access over search results, you can create security filters to trim results by identity, removing documents for which the requestor should not have access. For more information, see [Security filters](#) and [Secure with Active Directory](#).

See also

- [Role-based access control in Azure Cognitive Search](#)
- [Manage using PowerShell](#)
- [Performance and optimization article](#)

Configure IP firewall for Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

Azure Cognitive Search supports IP rules for inbound firewall support. This model provides an additional layer of security for your search service similar to the IP rules you'll find in an Azure virtual network security group. With these IP rules, you can configure your search service to be accessible only from an approved set of machines and/or cloud services. Access to data stored in your search service from these approved sets of machines and services will still require the caller to present a valid authorization token.

IMPORTANT

IP rules on your Azure Cognitive Search service can be configured using the Azure portal or the [Management REST API](#) version 2020-03-13.

Configure an IP firewall using the Azure portal

To set the IP access control policy in the Azure portal, go to your Azure Cognitive Search service page and select **Networking** on the navigation menu. Endpoint networking connectivity must be **Public**. If your connectivity is set to **Private**, you can only access your search service via a Private Endpoint.

The screenshot shows the Azure portal interface for managing an Azure Cognitive Search service named "azs-playground". The left sidebar has a red box around the "Networking" option under the "Settings" section. The main content area shows the "Networking" configuration page. At the top, there is a "Save" button with a red box around it. Below it, the "Endpoint network connectivity" section is visible, with a note about public endpoints and a "Learn more" link. Under "Endpoint connectivity (data)", the "Public" button is highlighted with a red box. The "Firewall" section allows adding IP ranges, with a note about client IP addresses and a "Learn more" link. An "Address range" input field is shown. The bottom of the page includes standard Azure portal footer links like "Properties", "Locks", and "Export template".

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.

NOTE

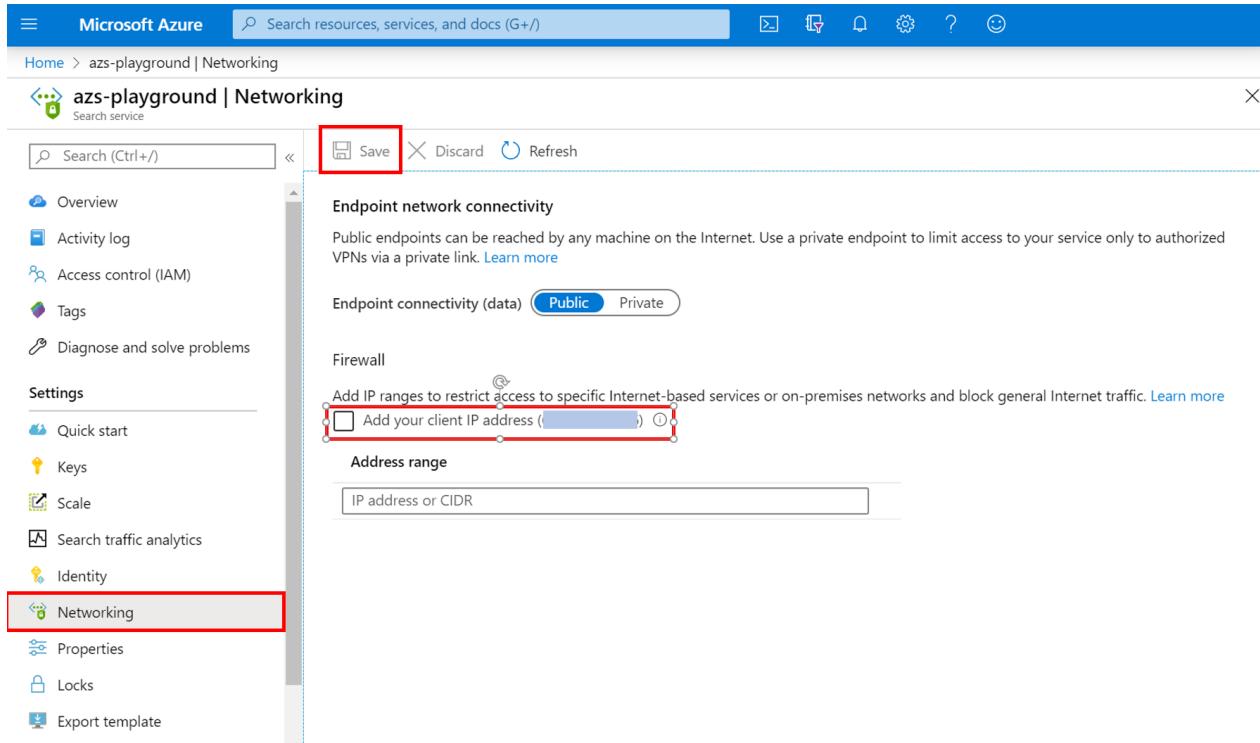
After you enable the IP access control policy for your Azure Cognitive Search service, all requests to the data plane from machines outside the allowed list of IP address ranges are rejected. When IP rules are configured, some features of the Azure portal are disabled. You'll be able to view and manage service level information, but portal access to index data and the various components in the service, such as the index, indexer, and skillset definitions, is restricted for security reasons.

Requests from your current IP

To simplify development, the Azure portal helps you identify and add the IP of your client machine to the allowed list. Apps running on your machine can then access your Azure Cognitive Search service.

The portal automatically detects your client IP address. It might be the client IP address of your machine or network gateway. Make sure to remove this IP address before you take your workload to production.

To add your current IP to the list of IPs, check **Add your client IP address**. Then select **Save**.



The screenshot shows the Azure portal interface for managing a Cognitive Search service named 'azs-playground'. The left sidebar lists various service management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Quick start, Keys, Scale, Search traffic analytics, Identity, and Networking. The 'Networking' option is highlighted with a red box. The main content area shows 'Endpoint network connectivity' settings. Under 'Endpoint connectivity (data)', there are two tabs: 'Public' (selected) and 'Private'. Below these tabs is a section for 'Firewall' which includes a note about adding IP ranges and a checkbox for 'Add your client IP address'. This checkbox is also highlighted with a red box. At the top right of the main content area, there are 'Save', 'Discard', and 'Refresh' buttons, with 'Save' being the one highlighted with a red box.

Troubleshoot issues with an IP access control policy

You can troubleshoot issues with an IP access control policy by using the following options:

Azure portal

Enabling an IP access control policy for your Azure Cognitive Search service blocks all requests from machines outside the allowed list of IP address ranges, including the Azure portal. You'll be able to view and manage service level information, but portal access to index data and the various components in the service, such as the index, indexer, and skillset definitions, is restricted for security reasons.

SDKs

When you access Azure Cognitive Search service using the SDK from machines that are not in the allowed list, a generic **403 Forbidden** response is returned with no additional details. Verify the allowed IP list for your account, and make sure that the correct configuration updated for your search service.

Next steps

For more information on accessing your search service via Private Link, see the following article:

- [Create a Private Endpoint for a secure connection to Azure Cognitive Search](#)

Create a Private Endpoint for a secure connection to Azure Cognitive Search

10/4/2020 • 6 minutes to read • [Edit Online](#)

In this article, you'll use the Azure portal to create a new Azure Cognitive Search service instance that can't be accessed via the internet. Next, you'll configure an Azure virtual machine in the same virtual network and use it to access the search service via a private endpoint.

Private endpoints are provided by [Azure Private Link](#), as a separate service. For more information about costs, see the [pricing page](#).

IMPORTANT

Private Endpoint support for Azure Cognitive Search can be configured using the Azure portal or the [Management REST API version 2020-03-13](#). When the service endpoint is private, some portal features are disabled. You'll be able to view and manage service level information, but portal access to index data and the various components in the service, such as the index, indexer, and skillset definitions, is restricted for security reasons.

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 enables 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 will 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 this information:

SETTING	VALUE
Subscription	Select your subscription
Resource group	Select Create new , enter <i>myResourceGroup</i> , then select OK

SETTING	VALUE
Name	Enter <i>MyVirtualNetwork</i>
Region	Select your desired region

- Leave the defaults for the rest of the settings. Click **Review + create** and then **Create**

Create a search service with a private endpoint

In this section, you will create a new Azure Cognitive Search service with a Private Endpoint.

- On the upper-left side of the screen in the Azure portal, select **Create a resource > Web > Azure Cognitive Search**.
- In **New Search Service - Basics**, enter or select this information:

SETTING	VALUE
PROJECT DETAILS	
Subscription	Select your subscription.
Resource group	Select myResourceGroup . You created this in the previous section.
INSTANCE DETAILS	
URL	Enter a unique name.
Location	Select your desired region.
Pricing tier	Select Change Pricing Tier and choose your desired service tier. (Not support on Free tier. Must be Basic or higher)

- Select **Next: Scale**.
- Leave the values as default and select **Next: Networking**.
- In **New Search Service - Networking**, select **Private** for **Endpoint connectivity(data)**.
- In **New Search Service - Networking**, select **+ Add** under **Private endpoint**.
- In **Create Private Endpoint**, enter or select this information:

SETTING	VALUE
Subscription	Select your subscription.
Resource group	Select myResourceGroup . You created this in the previous section.

SETTING	VALUE
Location	Select West US .
Name	Enter <i>myPrivateEndpoint</i> .
Target sub-resource	Leave the default searchService .
NETWORKING	
Virtual network	Select <i>MyVirtualNetwork</i> from resource group <i>myResourceGroup</i> .
Subnet	Select <i>mySubnet</i> .
PRIVATE DNS INTEGRATION	
Integrate with private DNS zone	Leave the default Yes .
Private DNS zone	Leave the default ** (New) <i>privatelink.search.windows.net</i> **.

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 just 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 this information:

SETTING	VALUE
PROJECT DETAILS	
Subscription	Select your subscription.
Resource group	Select myResourceGroup . You created this in the previous section.
INSTANCE DETAILS	
Virtual machine name	Enter <i>myVm</i> .

SETTING	VALUE
Region	Select West US or whatever region you are using.
Availability options	Leave the default No infrastructure redundancy required .
Image	Select Windows Server 2019 Datacenter .
Size	Leave the default Standard DS1 v2 .
ADMINISTRATOR ACCOUNT	
Username	Enter a username of your choosing.
Password	Enter a password of your choosing. 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	Leave the default Allow selected ports .
Select inbound ports	Leave the default RDP (3389) .
SAVE MONEY	
Already have a Windows license?	Leave the default No .

3. Select **Next: Disks**.

4. In **Create a virtual machine - Disks**, leave the defaults and select **Next: Networking**.

5. In **Create a virtual machine - Networking**, select this information:

SETTING	VALUE
Virtual network	Leave the default MyVirtualNetwork .
Address space	Leave the default 10.1.0.0/24 .
Subnet	Leave the default mySubnet (10.1.0.0/24) .
Public IP	Leave the default (new) myVm-ip .
Public inbound ports	Select Allow selected ports .
Select inbound ports	Select HTTP and RDP .

6. Select **Review + create**. You're taken to the **Review + create** page where Azure validates your

configuration.

- When you see the **Validation passed** message, select **Create**.

Connect to the VM

Download and then connect to the VM *myVm* as follows:

- In the portal's search bar, enter *myVm*.
- Select the **Connect** button. After selecting the **Connect** button, **Connect to virtual machine** opens.
- Select **Download RDP File**. Azure creates a Remote Desktop Protocol (.rdp) file and downloads it to your computer.
- Open the downloaded.rdp* file.
 - If prompted, select **Connect**.
 - 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.

- Select **OK**.
- You may receive a certificate warning during the sign-in process. If you receive a certificate warning, select **Yes** or **Continue**.
- Once the VM desktop appears, minimize it to go back to your local desktop.

Test connections

In this section, you will 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.

- In the Remote Desktop of *myVM*, open PowerShell.
- 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
```

- 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 in Postman using the REST API. Setting up requests from Postman requires the search service endpoint ([https://\[search service name\].search.windows.net](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 is not 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 does not exist, you have successfully configured a private endpoint for your search service.

Clean up resources

When you're done using the Private Endpoint, search service, and the VM, delete the resource group and all of the resources it contains:

1. Enter *myResourceGroup* in the **Search** box at the top of the portal and select *myResourceGroup* from the search results.
2. Select **Delete resource group**.
3. Enter *myResourceGroup* for **TYPE THE RESOURCE GROUP NAME** and 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?](#).

Security filters for trimming results in Azure Cognitive Search

10/4/2020 • 4 minutes to read • [Edit Online](#)

You can apply security filters to trim search results in Azure Cognitive Search based on user identity. This search experience generally requires comparing the identity of whoever requests the search against a field containing the principles who have permissions to the document. When a match is found, the user or principal (such as a group or role) has access to that document.

One way to achieve security filtering 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 simpler and faster approach is through the `search.in` function. If you use `search.in(Id, 'id1, id2, ...')` instead of an equality expression, you can expect sub-second response times.

This article shows you how to accomplish security filtering using the following steps:

- Create a field that contains the principal identifiers
- Push or update existing documents with the relevant principal identifiers
- Issue a search request with `search.in` filter

NOTE

The process of retrieving the principal identifiers is not covered in this document. You should get it from your identity service provider.

Prerequisites

This article assumes you have an [Azure subscription](#), an [Azure Cognitive Search service](#), and an [index](#).

Create security field

Your documents must include a field specifying which groups have access. This information becomes the filter criteria against which documents are selected or rejected from the result set returned to the issuer. Let's assume that we have an index of secured files, and each file is accessible by a different set of users.

1. Add field `group_ids` (you can choose any name here) as a `Collection(Edm.String)`. Make sure the field 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.

Make sure the field's `retrievable` attribute is set to `false` so that it is not returned as part of the search request.

2. Also add `file_id` and `file_name` fields for the sake of this example.

```
{  
    "name": "securedfiles",  
    "fields": [  
        {"name": "file_id", "type": "Edm.String", "key": true, "searchable": false, "sortable": false,  
        "facetable": false},  
        {"name": "file_name", "type": "Edm.String"},  
        {"name": "group_ids", "type": "Collection(Edm.String)", "filterable": true, "retrievable":  
        false}  
    ]  
}
```

Pushing 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 added:

```
POST https://[search service].search.windows.net/indexes/securedfiles/docs/index?api-version=2020-06-30  
Content-Type: application/json  
api-key: [admin key]
```

In the request body, specify the content of your documents:

```
{  
    "value": [  
        {  
            "@search.action": "upload",  
            "file_id": "1",  
            "file_name": "secured_file_a",  
            "group_ids": ["group_id1"]  
        },  
        {  
            "@search.action": "upload",  
            "file_id": "2",  
            "file_name": "secured_file_b",  
            "group_ids": ["group_id1", "group_id2"]  
        },  
        {  
            "@search.action": "upload",  
            "file_id": "3",  
            "file_name": "secured_file_c",  
            "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:

```
{  
    "value": [  
        {  
            "@search.action": "mergeOrUpload",  
            "file_id": "3",  
            "group_ids": ["group_id7", "group_id8", "group_id9"]  
        }  
    ]  
}
```

For full details on adding or updating documents, you can read [Edit documents](#).

Apply the security filter

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](#). Note that this sample shows how to search documents using a POST request.

Issue the HTTP POST request:

```
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:

```
{
  "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.

```
{
  [
    {
      "@search.score":1.0,
      "file_id":"1",
      "file_name":"secured_file_a",
    },
    {
      "@search.score":1.0,
      "file_id":"2",
      "file_name":"secured_file_b"
    }
  ]
}
```

Conclusion

This is how you can filter results based on user identity and Azure Cognitive Search `search.in()` function. You can use this function to pass in principle 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.

See also

- [Active Directory identity-based access control using Azure Cognitive Search filters](#)
- [Filters in Azure Cognitive Search](#)
- [Data security and access control in Azure Cognitive Search operations](#)

Security filters for trimming Azure Cognitive Search results using Active Directory identities

10/4/2020 • 5 minutes to read • [Edit Online](#)

This article demonstrates how to use Azure Active Directory (AAD) 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 AAD 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 AAD administrator permissions, required in this walkthrough for creating users, groups, and associations in AAD.

Your application must also be registered with AAD, as described in the following procedure.

Register your application with AAD

This step integrates your application with AAD for the purpose of accepting sign-ins of user and group accounts. If you are not an AAD admin in your organization, you might need to [create a new tenant](#) to perform the following steps.

1. Go to the [Application Registration Portal](#) > Converged app > Add an app.
2. Enter a name for your application, then click **Create**.
3. Select your newly registered application in the My Applications page.
4. On the application registration page > **Platforms** > **Add Platform**, choose **Web API**.
5. Still on the application registration page, go to > **Microsoft Graph Permissions** > **Add**.
6. In Select Permissions, add the following delegated permissions and then click **OK**:
 - **Directory.ReadWrite.All**
 - **Group.ReadWrite.All**
 - **User.ReadWrite.All**

Microsoft Graph provides an API that allows programmatic access to AAD 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 are adding search to an established application, you might have existing user and group identifiers in AAD. 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 AAD Group

```
// Instantiate graph client
GraphServiceClient graph = new GraphServiceClient(new DelegateAuthenticationProvider(...));
Group group = new Group()
{
    DisplayName = "My First Prog Group",
    SecurityEnabled = true,
    MailEnabled = false,
    MailNickname = "group1"
};
Group newGroup = await graph.Groups.Request().AddAsync(group);
```

Step 2: Create AAD User

```
User user = new User()
{
    GivenName = "First User",
    Surname = "User1",
    MailNickname = "User1",
    DisplayName = "First User",
    UserPrincipalName = "User1@FirstUser.com",
    PasswordProfile = new PasswordProfile() { Password = "*****" },
    AccountEnabled = true
};
User newUser = await graph.Users.Request().AddAsync(user);
```

Step 3: Associate user and group

```
await graph.Groups[newGroup.Id].Members.References.Request().AddAsync(newUser);
```

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 AAD. You can use [AAD 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 does not 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:

```
var actions = new IndexAction<SecuredFiles>[]  
    {  
        IndexAction.Upload(  
            new SecuredFiles()  
            {  
                FileId = "1",  
                Name = "secured_file_a",  
                GroupIds = new[] { groups[0] }  
            }),  
        ...  
    };  
  
var batch = IndexBatch.New(actions);  
  
_indexClient.Documents.Index(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 field 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 were not already cached, or the cache has expired, issue the [groups](#) request

```

private static void RefreshCacheIfRequired(string user)
{
    if (!_groupsCache.ContainsKey(user))
    {
        var groups = GetGroupIdsForUser(user).Result;
        _groupsCache[user] = groups;
    }
}

private static async Task<List<string>> GetGroupIdsForUser(string userPrincipalName)
{
    List<string> groups = new List<string>();
    var allUserGroupsRequest = graph.Users[userPrincipalName].GetMemberGroups(true).Request();

    while (allUserGroupsRequest != null)
    {
        IDirectoryObjectGetMemberGroupsRequestBuilder allUserGroups = await allUserGroupsRequest.PostAsync();
        groups = allUserGroups.ToList();
        allUserGroupsRequest = allUserGroups.NextPageRequest;
    }
    return groups;
}

```

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.

```

string filter = String.Format("groupIds/any(p:search.in(p, '{0}'))", string.Join(", ", groups.Select(g => g.ToString())));
SearchParameters parameters = new SearchParameters()
{
    Filter = filter,
    Select = new[] { "application essays" }
};

DocumentSearchResult<SecuredFiles> results = _indexClient.Documents.Search<SecuredFiles>("*", parameters);

```

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.

Conclusion

In this walkthrough, you learned techniques for using AAD sign-ins to filter documents in Azure Cognitive Search results, trimming the results of documents that do not match the filter provided on the request.

See also

- [Identity-based access control using Azure Cognitive Search filters](#)
- [Filters in Azure Cognitive Search](#)
- [Data security and access control in Azure Cognitive Search operations](#)

Setting up IP firewall rules to enable indexer access

10/4/2020 • 2 minutes to read • [Edit Online](#)

IP firewall rules on Azure resources such as storage accounts, Cosmos DB accounts, and Azure SQL servers only permit traffic originating from specific IP ranges to access data.

This article will describe how to configure the IP rules, via Azure portal, for a storage account so that Azure Cognitive Search indexers can access the data securely. While specific to storage, this guide can be directly translated to other Azure resources that also offer IP firewall rules for securing access to data.

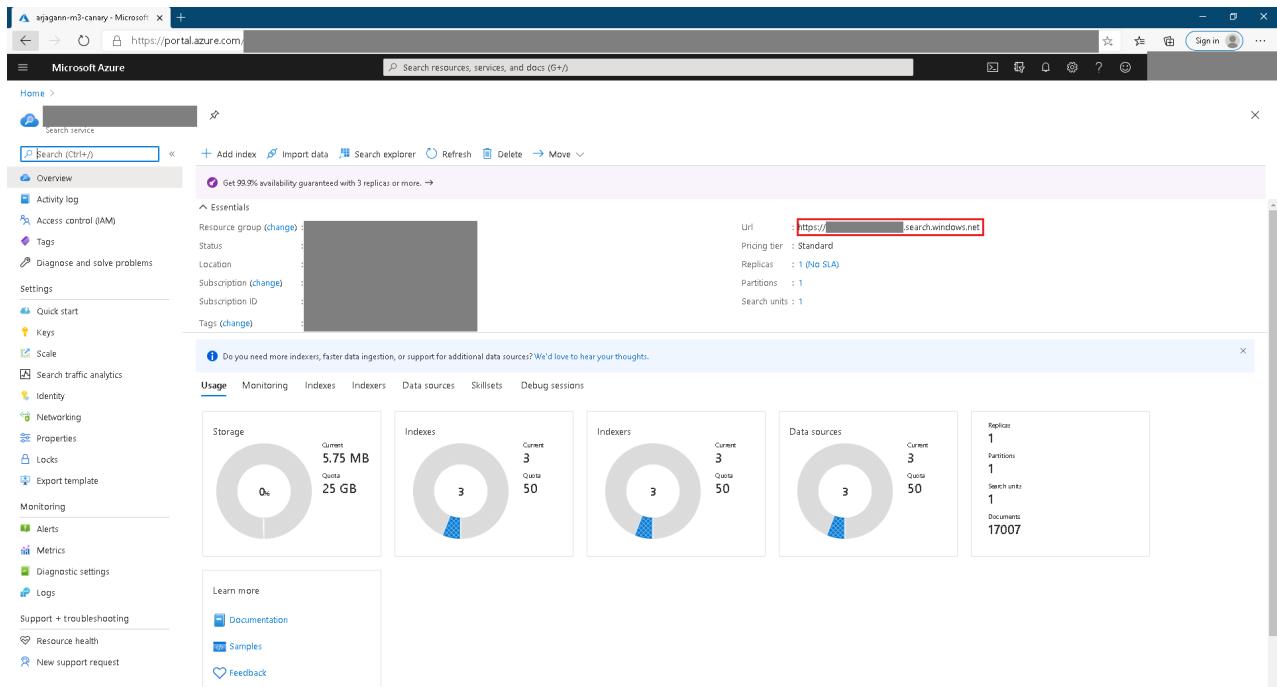
NOTE

IP firewall rules for storage account are only effective if the storage account and the search service are in different regions. If your setup does not permit this, we recommend utilizing the [trusted service exception option](#).

Get the IP address of the search service

Obtain the fully qualified domain name (FQDN) of your search service. This will look like

`<search-service-name>.search.windows.net`. You can find out the FQDN by looking up your search service on the Azure portal.



The screenshot shows the Azure portal interface for a search service named "Search service". The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Keys, Scale, Search traffic analytics, Identity, Networking, Properties, Export template, Monitoring, Alerts, Metrics, Diagnostic settings, Logs, Support + troubleshooting, Resource health, and New support request. The main content area displays the "Overview" tab for the search service. It shows the URL as `https://[REDACTED].search.windows.net`, with details: Pricing tier: Standard, Replicas: 1 (No SLA), Partitions: 1, and Search units: 1. Below this, there are four circular dashboards: Storage (Current: 5.75 MB, Quota: 25 GB), Indexes (Current: 3, Quota: 50), Indexers (Current: 3, Quota: 50), and Data sources (Current: 3, Quota: 50). A summary section indicates 1 replica, 1 partition, 1 search unit, and 1 document (17007 total).

The IP address of the search service can be obtained by performing a `nslookup` (or a `ping`) of the FQDN. This will be one of the IP addresses to add to the firewall rules.

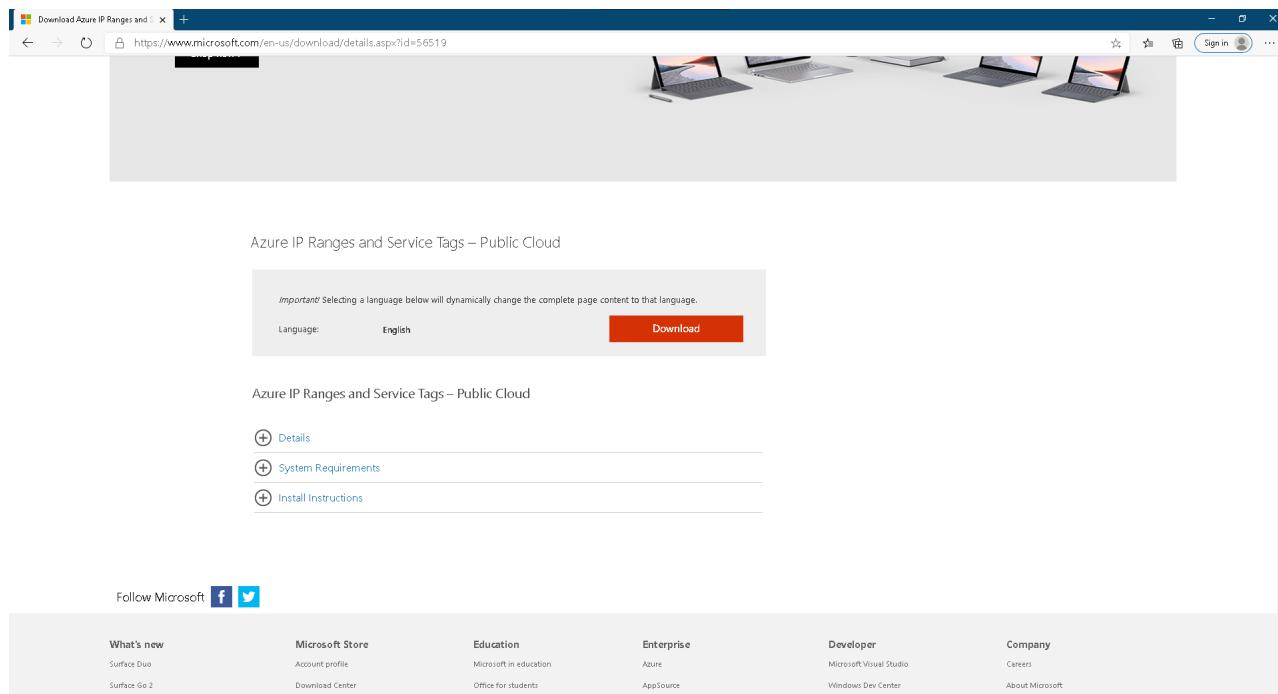
```
nslookup contoso.search.windows.net
Server: server.example.org
Address: 10.50.10.50

Non-authoritative answer:
Name: <name>
Address: 150.0.0.1
Aliases: contoso.search.windows.net
```

Get the IP address ranges for "AzureCognitiveSearch" service tag

The IP address ranges for the `AzureCognitiveSearch` service tag can be either obtained via the [discovery API \(preview\)](#) 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.



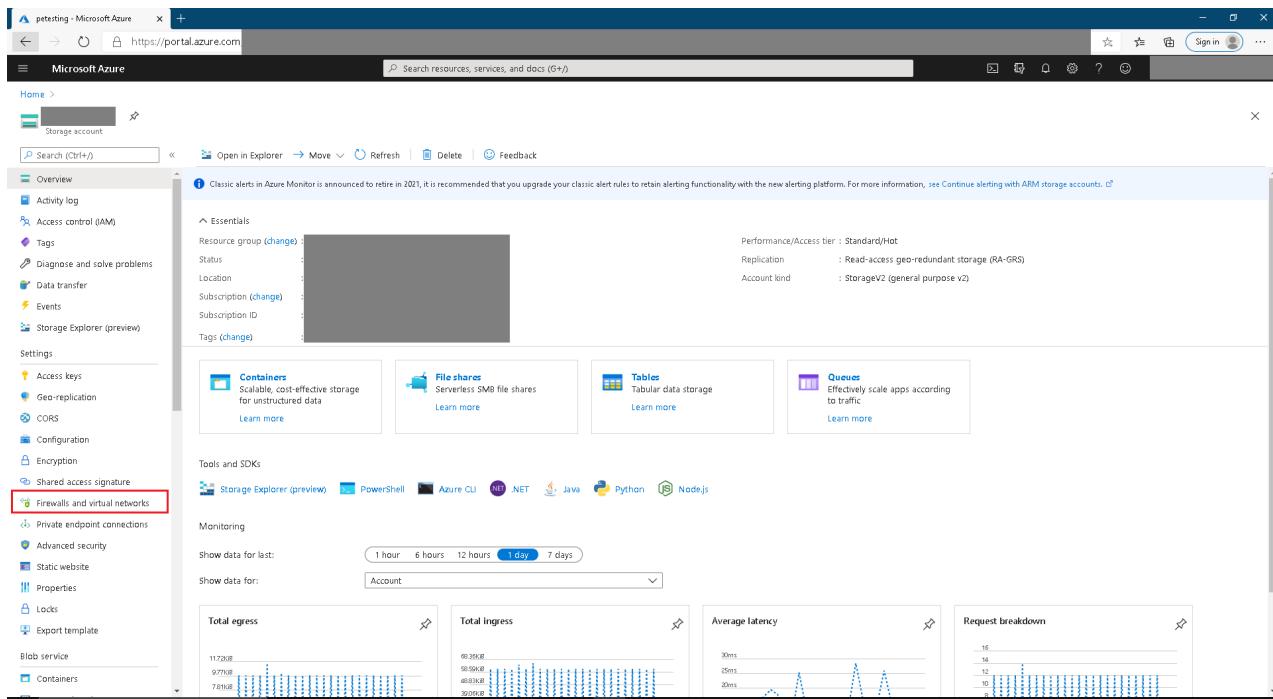
From the JSON file, assuming the search service is in West Central US, the list of IP addresses for the multi-tenant indexer execution environment are listed below.

```
{
  "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"
    ]
  }
}
```

For /32 IP addresses, drop the "/32" (52.253.133.74/32 -> 52.253.133.74), others can be used verbatim.

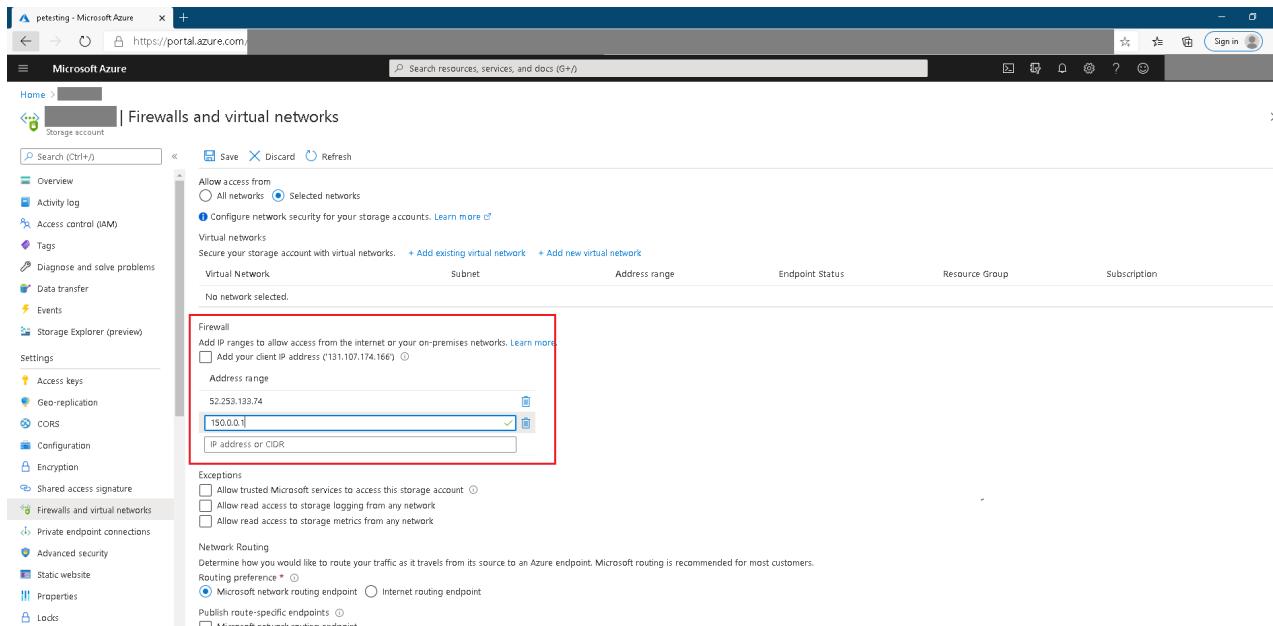
Add the IP address ranges to IP firewall rules

The easiest way to add IP address ranges to a storage account's firewall rule is via the Azure portal. Locate the storage account on the portal and navigate to the "Firewalls and virtual networks" tab.



The screenshot shows the Azure portal interface for a storage account. The left sidebar contains various navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), Settings, and Firewalls and virtual networks (which is highlighted with a red box). The main content area displays basic account details such as Resource group, Status, Location, Subscription, and Tags. It also features sections for Containers, File shares, Tables, and Queues, along with monitoring tools and SDKs. A chart at the bottom shows network traffic metrics over time.

Add the three IP addresses obtained previously (1 for the search service IP, 2 for the `AzureCognitiveSearch` service tag) in the address range and click "Save"



The screenshot shows the 'Firewalls and virtual networks' configuration page for a storage account. The left sidebar includes options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), Settings, and Firewalls and virtual networks. The main area shows a table for virtual networks, with a row for 'No network selected'. Below this, there's a 'Firewall' section where users can add IP ranges. An input field for 'Address range' contains the value '130.0.1' (highlighted with a red box). Other options in this section include 'Add your client IP address (131.107.174.166)' and 'IP address or CIDR'. There are also sections for 'Exceptions' and 'Network Routing'.

The firewall rules take 5-10 minutes to get updated after which indexers will be able to access the data in the storage account.

Next Steps

Now that you know how to get the two sets of IP addresses to allow access for indexes, use the following links to update the IP firewall rules for some common data sources.

- [Configure Azure Storage firewalls](#)
- [Configure IP firewall for CosmosDB](#)
- [Configure IP firewall for Azure SQL server](#)

Accessing data in storage accounts securely via trusted service exception

10/4/2020 • 2 minutes to read • [Edit Online](#)

Indexers that access data in storage accounts can make use of the [trusted service exception](#) capability to securely access data. This mechanism offers customers who are unable to grant [indexer access via IP firewall rules](#) a simple, secure, and free alternative to access data in storage accounts.

NOTE

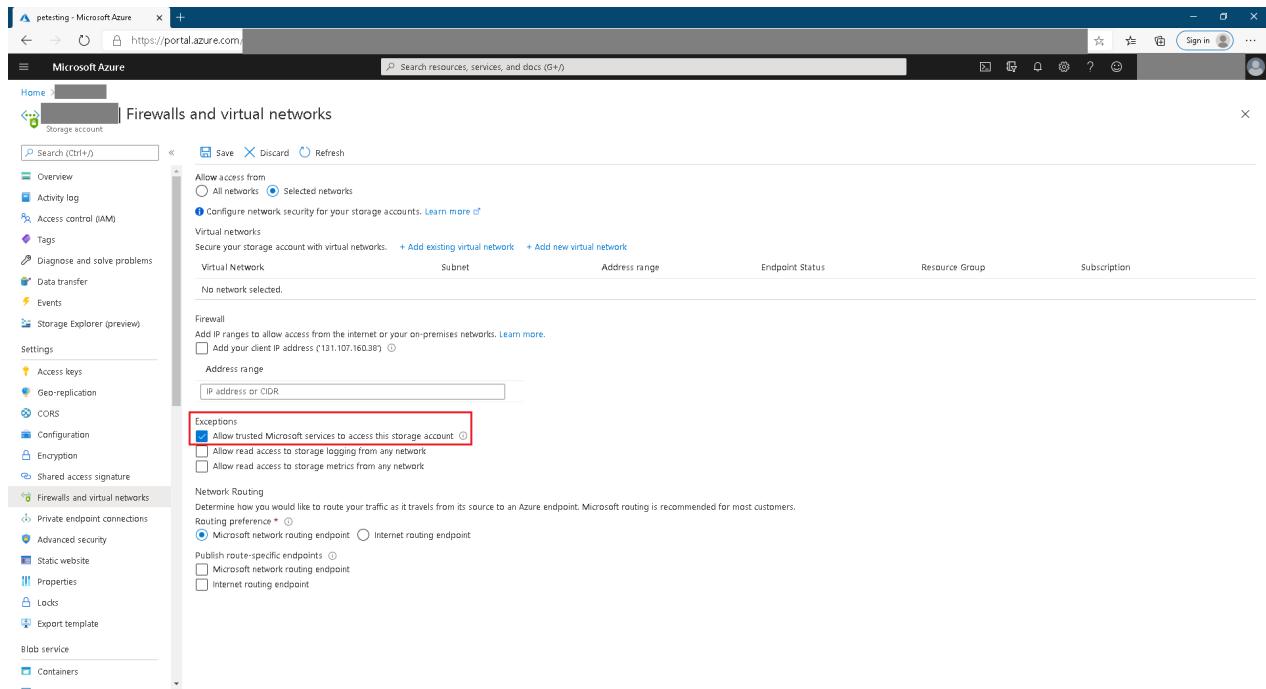
Support for accessing data in storage accounts via a trusted service exception is limited to Azure Blob storage and Azure Data Lake Gen2 storage. Azure table storage is not supported.

Step 1: Configure connection to the storage account via identity

Follow the details outlined in [the managed identity access guide](#) to configure indexers to access storage accounts via the search service's managed identity.

Step 2: Allow trusted Microsoft services to access the storage account

In the Azure portal, navigate to the "Firewalls and Virtual Networks" tab of the storage account. Ensure that the option "Allow trusted Microsoft services to access this storage account" is checked. 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.



The screenshot shows the Azure portal interface for managing a storage account. The left sidebar lists various storage account settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), Settings, and Firewalls and virtual networks. The main content area is titled 'Firewalls and virtual networks' and shows the 'Exceptions' section. In the 'Exceptions' section, the checkbox 'Allow trusted Microsoft services to access this storage account' is checked. Other options like 'Allow read access to storage logging from any network' and 'Allow read access to storage metrics from any network' are also listed but not checked. Below the exceptions, there are sections for 'Network Routing' and 'Publish route-specific endpoints'.

Indexers will now be able to access data in the storage account, even if the account is secured via IP firewall rules.

Next steps

Learn more about Azure Storage indexers:

- Azure Blob indexer
- Azure Data Lake Storage Gen2 indexer
- Azure Table indexer

Accessing secure resources via private endpoints

10/4/2020 • 7 minutes to read • [Edit Online](#)

Azure resources (such as storage accounts that are used as data sources), can be configured such that they can only be accessed from a specific list of virtual networks. They can also be configured to disallow any "public network" access. Customers can request Azure Cognitive Search to create an (outbound) [private endpoint connection](#) in order to securely access data from such data sources via indexers.

Shared Private Link Resources Management APIs

Private endpoints that are created by Azure Cognitive Search upon customer request, to access "secure" resources are referred to as *shared private link resources*. The customer is "sharing" access to a resource (such as a storage account), that has on-boarded to the [Azure Private Link service](#).

Azure Cognitive Search offers via the search management API, the ability to [Create or Update shared private link resources](#). You will use this API along with other *shared private link resources* management APIs to configure access to a secure resource from an Azure Cognitive Search indexer.

Private endpoint connections to some resources can only be created via the preview version of the search management API (`2020-08-01-Preview`), indicated with the "preview" tag in the table below. Resources without "preview" tag can be created via both the preview API as well as the GA API (`2020-08-01`)

The following are the list of Azure resources to which outbound private endpoints can be created from Azure Cognitive Search. `groupId` listed in the table below needs to be used exactly (case-sensitive) in the API to create a shared private link resource.

AZURE RESOURCE	GROUP ID
Azure Storage - Blob (or) ADLS Gen 2	<code>blob</code>
Azure Storage - Tables	<code>table</code>
Azure Cosmos DB - SQL API	<code>sql</code>
Azure SQL Database	<code>sqlServer</code>
Azure Database for MySQL (preview)	<code>mysqlServer</code>
Azure Key Vault	<code>vault</code>
Azure Functions (preview)	<code>sites</code>

The list of Azure resources for which outbound private endpoint connections are supported can also be queried via the [List Supported API](#).

For the purposes of this guide, a mix of [ARMClient](#) and [Postman](#) are used to demonstrate the REST API calls.

NOTE

Throughout this guide, let's assume that the name of the search service is **contoso-search** which exists in the resource group **contoso** of a subscription with subscription ID **00000000-0000-0000-0000-000000000000**. The resource ID of this search service will be

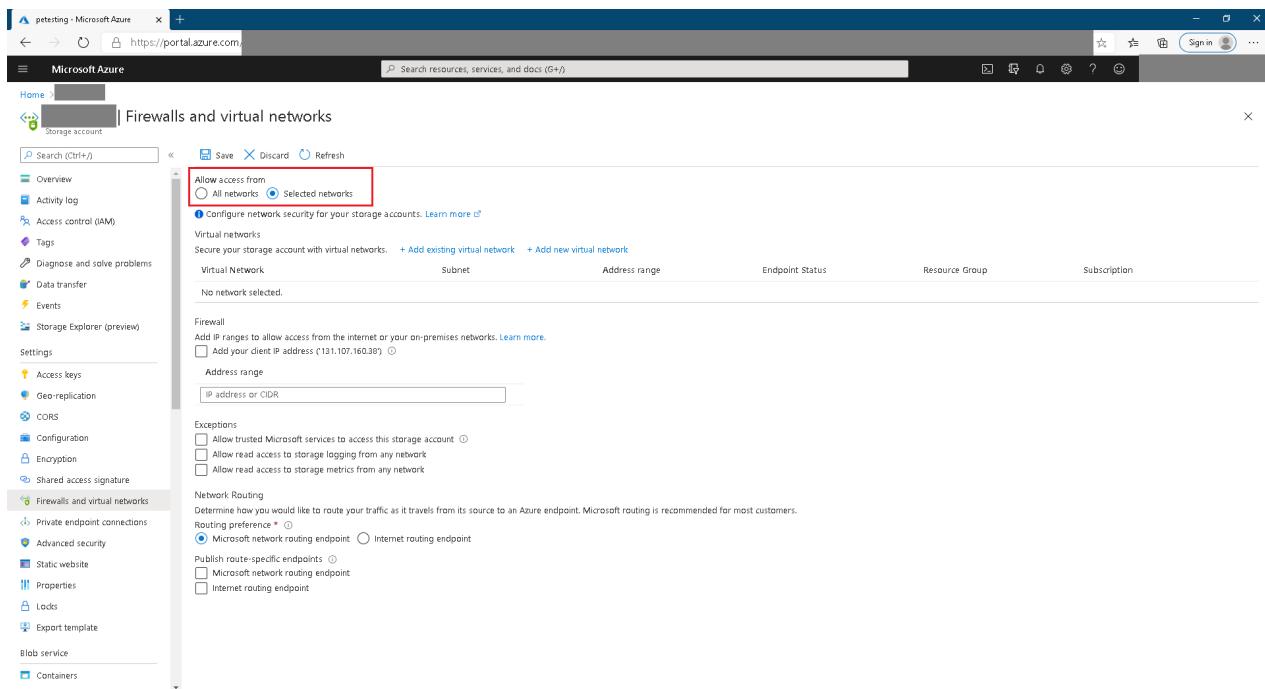
```
/subscriptions/00000000-0000-0000-0000-  
000000000000/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-search
```

The rest of the guide will show how the **contoso-search** service can be configured so that its indexers can access data from the secure storage account

```
/subscriptions/00000000-0000-0000-0000-  
000000000000/resourceGroups/contoso/providers/Microsoft.Storage/storageAccounts/contoso-storage
```

Securing your storage account

Configure the storage account to [allow access only from specific subnets](#). Via the Azure portal, if you check this option and leave the set empty, it means that no traffic from any virtual network is allowed.



The screenshot shows the Azure portal interface for managing a storage account. On the left, there's a sidebar with various navigation options like Home, Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Data transfer, Events, Storage Explorer (preview), Settings, Access keys, Geo-replication, CORS, Configuration, Encryption, Shared access signature, Firewalls and virtual networks, Private endpoint connections, Advanced security, Static website, Properties, Logs, Export template, Blob service, and Containers. The 'Firewalls and virtual networks' section is currently selected. In the main content area, there's a form for configuring network security. It includes fields for 'Virtual networks' (with options to 'Secure your storage account with virtual networks', 'Add existing virtual network', and 'Add new virtual network'), 'Virtual Network' (with a dropdown for 'Subnet' and 'Address range'), 'Endpoint Status', 'Resource Group', and 'Subscription'. Below these are sections for 'Firewall' (with an 'Add IP ranges...' button and a note about adding client IP addresses) and 'Address range' (with an 'IP address or CIDR' input field). At the top of the configuration pane, there are buttons for 'Save', 'Discard', and 'Refresh'. A red box highlights the 'Selected networks' radio button under the 'Allow access from' section, which is currently selected over 'All networks'.

NOTE

The [trusted Microsoft service approach](#) can be used to bypass virtual network or IP restrictions on such a storage account and can enable the search service to access data in the storage account as described in the [how to guide](#). However, when using this approach communication between Azure Cognitive Search and the storage account happens via the public IP address of the storage account, over the secure Microsoft backbone network.

Step 1: Create a shared private link resource to the storage account

Make the following API call to request Azure Cognitive Search to create an outbound private endpoint connection to the storage account

```
armclient PUT https://management.azure.com/subscriptions/00000000-0000-0000-0000-  
000000000000/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-  
search/sharedPrivateLinkResources/blob-pe?api-version=2020-08-01 create-pe.json
```

The contents of `create-pe.json` file (that represents the request body to the API) are as follows:

```
{
  "name": "blob-pe",
  "properties": {
    "privateLinkResourceId": "/subscriptions/00000000-0000-0000-0000-
0000000000/resourceGroups/contoso/providers/Microsoft.Storage/storageAccounts/contoso-storage",
    "groupId": "blob",
    "requestMessage": "please approve"
  }
}
```

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 -

1. A private endpoint allocated with a private IP address in a `"Pending"` state. The private IP address is obtained from the address space allocated to the virtual network of the search service specific private indexer execution environment. Upon approval of the private endpoint, any communication from Azure Cognitive Search to the storage account originates from the private IP address and a secure private link channel.
2. A private DNS zone for the type of resource, based on the `groupId`. This will ensure that any DNS lookup to the private resource utilizes the IP address associated with the private endpoint.

Make sure to specify the correct `groupId` for the type of resource for which you are creating the private endpoint. Any mismatch will result in a non-successful response message.

Like all asynchronous Azure operations, the `PUT` call returns a `Azure-AsyncOperation` header value that will look as follows:

```
"Azure-AsyncOperation": "https://management.azure.com/subscriptions/00000000-0000-0000-0000-
0000000000/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-
search/sharedPrivateLinkResources/blob-pe/operationStatuses/08586060559526078782?api-version=2020-08-01"
```

This URI can be polled periodically to obtain the status of the operation. We recommend waiting until the shared private link resource operation status has reached a terminal state (that is, `succeeded`) before proceeding.

```
armclient GET https://management.azure.com/subscriptions/00000000-0000-0000-0000-
0000000000/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-
search/sharedPrivateLinkResources/blob-pe/operationStatuses/08586060559526078782?api-version=2020-08-01"
```

```
{
  "status": "running" | "succeeded" | "failed"
}
```

Step 2a: Approve the private endpoint connection for the storage account

NOTE

This section uses Azure portal to walk through the approval flow for a private endpoint to storage. The [REST API](#) available via storage resource provider (RP) can also be used instead.

Other providers such as CosmosDB, Azure SQL server etc., also offer similar RP APIs to manage private endpoint connections.

Navigate to the "Private endpoint connections" tab of the storage account on Azure portal. There should be a request for a private endpoint connection, with the request message from the previous API call (once the asynchronous operation has `succeeded`).

Connection name	Connection state	Private endpoint	Description
blob-pe	Approved	blob-pe	please approve

Select the private endpoint that was created by Azure Cognitive Search (use the "Private endpoint" column to identify the private endpoint connection by the name specified in the previous API) and choose "Approve", with an appropriate message (the message isn't significant). Make sure the private endpoint connection appears as follows (it could anywhere from 1-2 minutes for the status to be updated on the portal)

Connection name	Connection state	Private endpoint	Description
blob-pe	Approved	blob-pe	please approve

After the private endpoint connection request is approved, it means that traffic is *capable* of flowing through the private endpoint. Once the private endpoint is approved Azure Cognitive Search will create the necessary DNS zone mappings in the DNS zone created for it.

Step 2b: Query the status of the shared private link resource

To confirm that the shared private link resource has been updated after approval, obtain its status via the [GET API](#).

```
armclient GET https://management.azure.com/subscriptions/00000000-0000-0000-0000-000000000000/resourceGroups/contoso/providers/Microsoft.Search/searchServices/contoso-search/sharedPrivateLinkResources/blob-pe?api-version=2020-08-01
```

If the `properties.provisioningState` of the resource is `Succeeded` and `properties.status` is `Approved`, it means

that the shared private link resource is functional and indexers can be configured to communicate over the private endpoint.

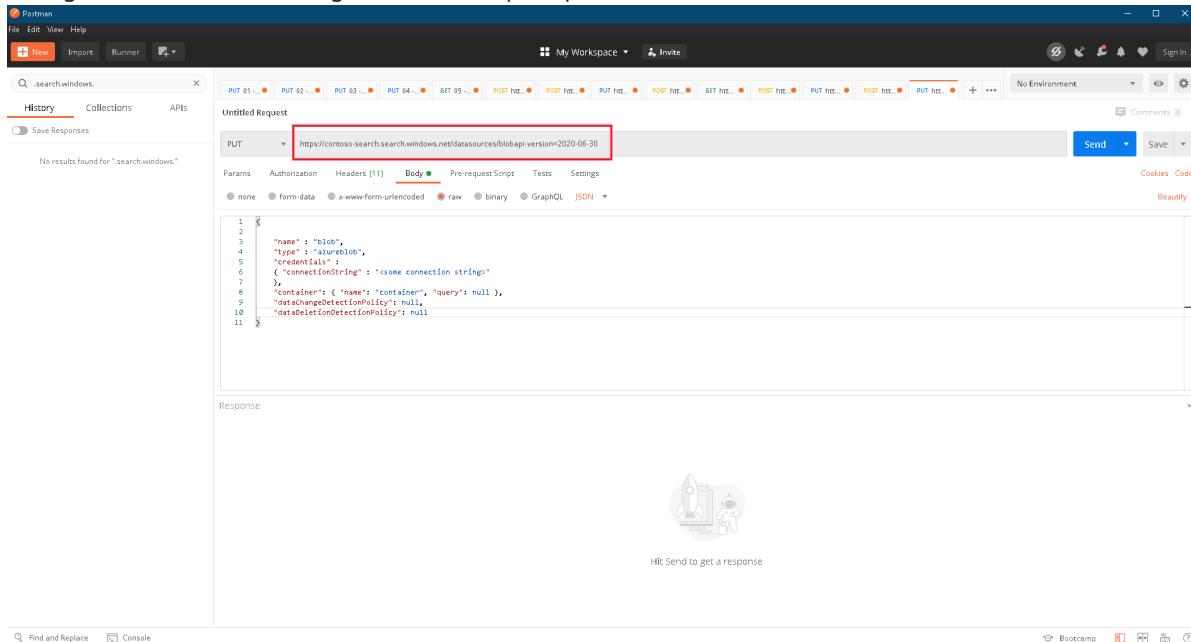
```
{  
    "name": "blob-pe",  
    "properties": {  
        "privateLinkResourceId": "/subscriptions/00000000-0000-0000-0000-  
0000000000/resourceGroups/contoso/providers/Microsoft.Storage/storageAccounts/contoso-storage",  
        "groupId": "blob",  
        "requestMessage": "please approve",  
        "status": "Approved",  
        "resourceRegion": null,  
        "provisioningState": "Succeeded"  
    }  
}
```

Step 3: Configure indexer to run in the private environment

NOTE

This step can be performed even before the private endpoint connection is approved. Until the private endpoint connection is approved, any indexer that tries to communicate with a secure resource (such as the storage account), will end up in a transient failure state. New indexers will fail to be created. As soon as the private endpoint connection is approved, indexers will be able to access the private storage account.

1. [Create a data source](#) that points to the secure storage account and an appropriate container within the storage account. The following shows this request performed via Postman.



2. Similarly [create an index](#) and optionally [create a skillset](#) using the REST API.
3. [Create an indexer](#) that points to the data source, index, and skillset created above. In addition, force the indexer to run in the private execution environment, by setting the indexer configuration property `executionEnvironment` to `"Private"`.

The screenshot shows the Postman interface with a PUT request to the specified URL. The JSON body contains the indexer configuration, with the execution environment set to "private".

```
1 {
2     "name": "Indexer",
3     "dataSourceId": null,
4     "dataConnectionString": null,
5     "targetIndexName": "Index1",
6     "disabled": null,
7     "scheduled": null,
8     "configuration": [
9         {
10            "configuration": {
11                "executionEnvironment": "private"
12            }
13        }
14    ],
15    "fieldMappings": []
}
```

The indexer should be created successfully, and should be making progress - indexing content from the storage account over the private endpoint connection. The status of the indexer can be monitored via the [Indexer status API](#).

NOTE

If you already have existing indexers, you can simply update them via the [PUT API](#) to set the `executionEnvironment` to `"Private"`.

Troubleshooting issues

- When creating an indexer, if creation fails with an error message similar to "Data source credentials are invalid", it means that either the private endpoint connection has not been *Approved* or it is not functional. Obtain the status of the shared private link resource using the [GET API](#). If it has been *Approved* check the `properties.provisioningState` of the resource. If it is `Incomplete`, this means some of the underlying dependencies for the resource failed to provision - reissue the `PUT` request to "re-create" the shared private link resource that should fix the issue. A re-approval might be necessary - check the status of the resource once again to verify.
- If the indexer is created without setting its `executionEnvironment`, the indexer creation might succeed, but its execution history will show that indexer runs are unsuccessful. You should [update the indexer](#) to specify the execution environment.
- If the indexer is created without setting the `executionEnvironment` and it runs successfully, it means that Azure Cognitive Search has decided that its execution environment is the search service specific "private" environment. However, this can change based on a variety of factors (resources consumed by the indexer, the load on the search service, and so on) and can fail at a later point - we highly recommend you set the `executionEnvironment` as `"Private"` to ensure that it will not fail in the future.
- [Quotas and limits](#) determine how many shared private link resources can be created and depend on the SKU of the search service.

Next steps

Learn more about private endpoints:

- [What are private endpoints?](#)
- [DNS configurations needed for private endpoints](#)

Service administration for Azure Cognitive Search in the Azure portal

10/4/2020 • 8 minutes to read • [Edit Online](#)

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 service administration tasks that you can perform in the [Azure portal](#) for a search service that you've already provisioned. Service administration is lightweight by design, limited to the following tasks:

- Check storage using the mid-page **Usage** link.
- Check query volumes and latency using the mid-page **Monitoring** link, and whether requests were throttled.
- Manage access using the **Keys** page to the left.
- Adjust capacity using the **Scale** page to the left.

The same tasks performed in the portal can also be handled programmatically through the [Management APIs](#) and [Az.Search PowerShell module](#). Administrative tasks are fully represented across portal and programmatic interfaces. There is no specific administrative task that is available in only one modality.

Azure Cognitive Search leverages other Azure services for deeper monitoring and management. By itself, the only data stored with a search service is content (indexes, indexer and data source definitions, and other objects). Metrics reported out to portal pages are pulled from internal logs on a rolling 30-day cycle. For user-controlled log retention and additional events, you will need [Azure Monitor](#).

Fixed service properties

Several aspects of a search service are determined when the service is provisioned and cannot be changed later:

- Service name (you cannot rename a service)
- Service location (you cannot currently move an intact service to another region)
- Maximum replica and partition counts (determined by the tier, Basic or Standard)

If you started with Basic with its maximum of one partition, and you now need more partitions, you will need to [create a new service](#) at a higher tier and recreate your content on the new service.

Administrator rights

Provisioning or decommissioning the service itself can be done by an Azure subscription administrator or co-administrator.

Regarding access to the endpoint, anyone with access to the service URL and an api-key has access to content. For more information about keys, see [Manage the api-keys](#).

- Read-only access to the service is query rights, typically granted to a client application by giving it the URL and a query api-key.
- Read-write access provides the ability to add, delete, or modify server objects, including api-keys, indexes, indexers, data sources, and schedules. Read-write access is granted by giving the URL, an admin API key.

Rights to the service provisioning apparatus is granted through role assignments. [Azure role-based access control \(Azure RBAC\)](#) is an authorization system built on [Azure Resource Manager](#) for provisioning of Azure resources.

In the context of Azure Cognitive Search, [Azure role assignments](#) will determine who can perform tasks, regardless

of whether they are using the [portal](#), [PowerShell](#), or the [Management REST APIs](#):

- Create or delete a service
- Scale the service
- Delete or regenerate API keys
- Enable diagnostic logging (create services)
- Enable traffic analytics (create services)

TIP

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](#).

Logging and system information

At the Basic tier and above, Microsoft monitors all Azure Cognitive Search services for 99.9% availability per service level agreements (SLA). If the service is slow or request throughput falls below SLA thresholds, support teams review the log files available to them and address the issue.

Azure Cognitive Search leverages [Azure Monitor](#) to collect and store indexing and query activity. A search service by itself stores just its content (indexes, indexer definitions, data source definitions, skillset definitions, synonym maps). Caching and logged information is stored off-service, often in an Azure Storage account. For more information about logging indexing and query workloads, see [Collect and analyze log data](#).

In terms of general information about your service, using just the facilities built into Azure Cognitive Search itself, you can obtain information in the following ways:

- Using the service [Overview](#) page, through notifications, properties, and status messages.
- Using [PowerShell](#) or the [Management REST API](#) to [get service properties](#). There is no new information or operations provided at the programmatic layer. The interfaces exist so that you can write scripts.

Monitor resource usage

In the dashboard, resource monitoring is limited to the information shown in the service dashboard and a few metrics that you can obtain by querying the service. On the service dashboard, in the Usage section, you can quickly determine whether partition resource levels are adequate for your application. You can provision external resources, such as Azure monitoring, if you want to capture and persist logged events. For more information, see [Monitoring Azure Cognitive Search](#).

Using the search service REST API, you can get a count on documents and indexes programmatically:

- [Get Index Statistics](#)
- [Count Documents](#)

Disaster recovery and service outages

Although we can salvage your data, Azure Cognitive Search does not provide instant failover of the service if there is an outage at the cluster or data center level. If a cluster fails in the data center, the operations team will detect and work to restore service. You will experience downtime during service restoration, but you can request service credits to compensate for service unavailability per the [Service Level Agreement \(SLA\)](#).

If continuous service is required in the event of catastrophic failures outside of Microsoft's control, you could [provision an additional service](#) in a different region and implement 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 leveraging 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 are 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 re-point the indexer to the new primary replica.

If you do not use indexers, you would use your application code to push objects and data to different search services in parallel. For more information, see [Performance and optimization in Azure Cognitive Search](#).

Backup and restore

Because Azure Cognitive Search is not a primary data storage solution, we do not 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 backup 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.

Scale up or down

Every search service starts with a minimum of one replica and one partition. If you signed up for a [tier that supports more capacity](#), click **Scale** on the left navigation pane to adjust resource usage.

When you add capacity through either resource, the service uses them automatically. No further action is required on your part, but there is a slight delay before the impact of the new resource is realized. It can take 15 minutes or more to provision additional resources.

Add replicas

Increasing queries per second (QPS) or achieving high availability is done by adding replicas. Each replica has one copy of an index, so adding one more replica translates to one more index available for handling service query requests. A minimum of 3 replicas are required for high availability (see [Adjust capacity](#) for details).

A search service having more replicas can load balance query requests over a larger number of indexes. Given a level of query volume, query throughput is going to be faster when there are more copies of the index available to service the request. If you are experiencing query latency, you can expect a positive impact on performance once the additional replicas are online.

Although query throughput goes up as you add replicas, it does not precisely double or triple as you add replicas to your service. All search applications are subject to external factors that can impinge on query performance. Complex queries and network latency are two factors that contribute to variations in query response times.

Add partitions

It's more common to add replicas, but when storage is constrained, you can add partitions to get more capacity. The tier at which you provisioned the service determines whether partitions can be added. The Basic tier is locked at one partition. Standard tiers and above support additional partitions.

Partitions are added in divisors of 12 (specifically, 1, 2, 3, 4, 6, or 12). This is an artifact of sharding. An index is created in 12 shards, which can all be stored on 1 partition or equally divided into 2, 3, 4, 6, or 12 partitions (one shard per partition).

Remove replicas

After periods of high query volumes, you can use the slider to reduce replicas after search query loads have normalized (for example, after holiday sales are over). There are no further steps required on your part. Lowering

the replica count relinquishes virtual machines in the data center. Your query and data ingestion operations will now run on fewer VMs than before. The minimum requirement is one replica.

Remove partitions

In contrast with removing replicas, which requires no extra effort on your part, you might have some work to do if you are using more storage than can be reduced. For example, if your solution is using three partitions, downsizing to one or two partitions will generate an error if the new storage space is less than required for hosting your index. As you might expect, your choices are to delete indexes or documents within an associated index to free up space, or keep the current configuration.

There is no detection method that tells you which index shards are stored on specific partitions. Each partition provides approximately 25 GB in storage, so you will need to reduce storage to a size that can be accommodated by the number of partitions you have. If you want to revert to one partition, all 12 shards will need to fit.

To help with future planning, you might want to check storage (using [Get Index Statistics](#)) to see how much you actually used.

Next steps

- Automate with [PowerShell](#)
- Review [performance and optimization](#) techniques
- Review [security features](#) to protect content and operations
- Enable [diagnostic logging](#) to monitor query and indexing workloads

Manage your Azure Cognitive Search service with PowerShell

10/4/2020 • 6 minutes to read • [Edit Online](#)

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](#)
- [Regenerate admin API-keys](#)
- [Create or delete query api-keys](#)
- [Scale up or down with replicas and partitions](#)

Occasionally, questions are asked about tasks *not* on the above list. Currently, you cannot use either the **Az.Search** module or the management REST API to change a server name, region, or tier. Dedicated resources are allocated when a service is created. As such, changing the underlying hardware (location or node type) requires a new service. Similarly, there are no tools or APIs for transferring content, such as an index, from one service to another.

Within a service, content creation and management is through [Search Service REST API](#) or [.NET SDK](#). While there are no dedicated PowerShell commands for content, you can write PowerShell script that calls REST or .NET APIs to create and load indexes.

Check versions and load modules

The examples in this article are interactive and require elevated permissions. Azure PowerShell (the **Az** module) must be installed. For more information, see [Install Azure PowerShell](#).

PowerShell version check (5.1 or later)

Local PowerShell must be 5.1 or later, on any supported operating system.

```
$PSVersionTable.PSVersion
```

Load Azure PowerShell

If you aren't sure whether **Az** is installed, run the following command as a verification step.

```
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 instructions to see if you missed a step.

```
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.

```
Connect-AzAccount
```

If you hold multiple Azure subscriptions, set your Azure subscription. To see a list of your current subscriptions, run this command.

```
Get-AzSubscription | sort SubscriptionName | Select SubscriptionName
```

To specify the subscription, run the following command. In the following example, the subscription name is `ContosoSubscription`.

```
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.

```
Get-AzResource -ResourceType Microsoft.Search/searchServices | ft
```

From the list of services, return information about a specific resource.

```
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/<alpha-numeric-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.

```
Install-Module -Name Az.Search
```

List all Az.Search commands

As a verification step, return a list of commands provided in the module.

```
Get-Command -Module Az.Search
```

Results should look similar to the following output.

CommandType	Name	Version	Source
Cmdlet	Get-AzSearchAdminKeyPair	0.7.1	Az.Search
Cmdlet	Get-AzSearchQueryKey	0.7.1	Az.Search
Cmdlet	Get-AzSearchService	0.7.1	Az.Search
Cmdlet	New-AzSearchAdminKey	0.7.1	Az.Search
Cmdlet	New-AzSearchQueryKey	0.7.1	Az.Search
Cmdlet	New-AzSearchService	0.7.1	Az.Search
Cmdlet	Remove-AzSearchQueryKey	0.7.1	Az.Search
Cmdlet	Remove-AzSearchService	0.7.1	Az.Search
Cmdlet	Set-AzSearchService	0.7.1	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.

```
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](#).

```
New-AzSearchService -ResourceGroupName "demo-westus" -Name "my-demo-searchapp" -Sku "Standard" -Location
"West US" -PartitionCount 3 -ReplicaCount 3
```

Results should look similar to the following output.

```
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
```

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`.

```
New-AzSearchAdminKey -ResourceGroupName <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.

```
New-AzSearchQueryKey -ResourceGroupName <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.

```
Set-AzSearchService -ResourceGroupName <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
```

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](#)

Move your Azure Cognitive Search service to another Azure region

10/4/2020 • 2 minutes to read • [Edit Online](#)

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 achieving the same outcome.

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](#)

Set Azure roles for administrative access to Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

Azure provides a [global role-based authorization model](#) for all services managed through the portal or Resource Manager APIs. Owner, Contributor, and Reader roles determine the level of *service administration* for Active Directory users, groups, and security principals assigned to each role.

NOTE

There is no role-based access control (RBAC) for securing portions of an index or a subset of documents. For identity-based access over search results, you can create security filters to trim results by identity, removing documents for which the requestor should not have access. For more information, see [Security filters](#) and [Secure with Active Directory](#).

Management tasks by role

For Azure Cognitive Search, roles are associated with permission levels that support the following management tasks:

ROLE	TASK
Owner	<p>Create or delete the service or any object on the service, including api-keys, indexes, indexers, indexer data sources, and indexer schedules.</p> <p>View service status, including counts and storage size.</p> <p>Add or delete role membership (only an Owner can manage role membership).</p> <p>Subscription administrators and service owners have automatic membership in the Owners role.</p>
Contributor	<p>Same level of access as Owner, minus Azure role management. For example, a Contributor can create or delete objects, or view and regenerate api-keys, but cannot modify role memberships.</p>
Search Service Contributor built-in role	Equivalent to the Contributor role.
Reader	<p>View service essentials and metrics. Members of this role cannot view index, indexer, data source, or key information.</p>

Roles do not grant access rights to the service endpoint. Search service operations, such as index management, index population, and queries on search data, are controlled through api-keys, not roles. For more information, see [Manage api-keys](#).

Permissions table

The following table summarizes the operations allowed in Azure Cognitive Search and which key unlocks access a particular operation.

OPERATION	PERMISSIONS
Create a service	Azure subscription holder
Scale a service	Admin key, RBAC Owner, or Contributor on the resource
Delete a service	Admin key, RBAC Owner, or Contributor on the resource
Create, modify, delete objects on the service: Indexes and component parts (including analyzer definitions, scoring profiles, CORS options), indexers, data sources, synonyms, suggesters	Admin key, RBAC Owner, or Contributor on the resource
Query an index	Admin or query key (RBAC not applicable)
Query system information, such as returning statistics, counts, and lists of objects	Admin key, RBAC on the resource (Owner, Contributor, Reader)
Manage admin keys	Admin key, RBAC Owner or Contributor on the resource
Manage query keys	Admin key, RBAC Owner or Contributor on the resource

See also

- [Manage using PowerShell](#)
- [Performance and optimization in Azure Cognitive Search](#)
- [Get started with Role-Based Access Control in the Azure portal.](#)

Monitor operations and activity of Azure Cognitive Search

10/4/2020 • 5 minutes to read • [Edit Online](#)

This article is an overview of monitoring concepts and tools for Azure Cognitive Search. For holistic monitoring, you can use a combination of built-in functionality and add-on services like Azure Monitor.

Altogether, you can track the following:

- Service: health/availability and changes to service configuration.
- Storage: both used and available, with counts for each content type relative to the quota allowed for the service tier.
- Query activity: volume, latency, and throttled or dropped queries. Logged query requests require [Azure Monitor](#).
- Indexing activity: requires [diagnostic logging](#) with Azure Monitor.

A search service does not support per-user authentication, so no identity information will be found in the logs.

Built-in monitoring

Built-in monitoring refers to activities that are logged by a search service. With the exception of diagnostics, no configuration is required for this level of monitoring.

Azure Cognitive Search maintains internal data on a rolling 30-day schedule for reporting on service health and query metrics, which you can find in the portal or through these [REST APIs](#).

The following screenshot helps you locate monitoring information in the portal. Data becomes available as soon as you start using the service. Portal pages are refreshed every few minutes.

- **Monitoring** tab, on the main Overview page, shows query volume, latency, and whether the service is under pressure.
- **Activity log**, in the left navigation pane, is connected to Azure Resource Manager. The activity log reports on actions undertaken by Resource Manager: service availability and status, changes to capacity (replicas and partitions), and API key-related activities.
- **Monitoring** settings, further down, provides configurable alerts, metrics, and diagnostic logs. Create these when you need them. Once data is collected and stored, you can query or visualize the information for insights.

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 servicing 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.

APIs useful for monitoring

You can use the following APIs to retrieve the same information found in the Monitoring and Usage tabs in the portal.

- [GET Service Statistics](#)
- [GET Index Statistics](#)
- [GET Document Counts](#)
- [GET Indexer Status](#)

Activity logs and service health

The [Activity log](#) page in the portal collects information from Azure Resource Manager and reports on changes to service health. You can monitor the activity log for critical, error, and warning conditions related to service health.

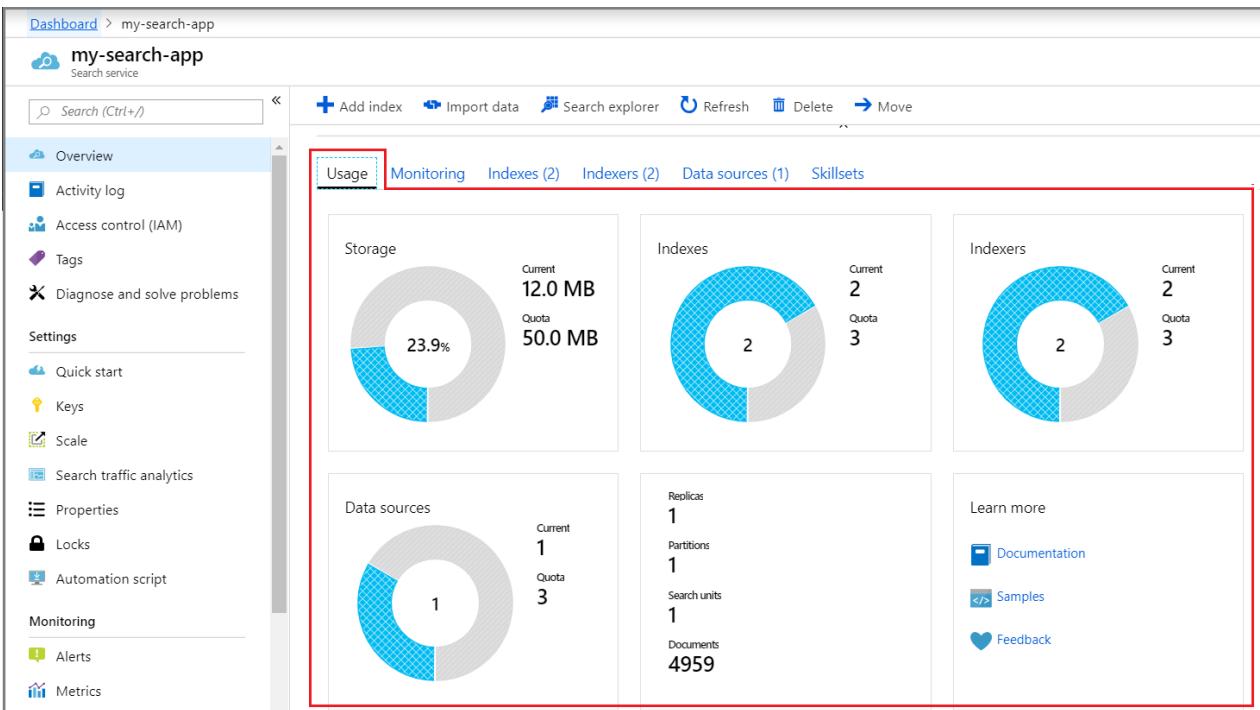
Common entries include references to API keys - generic informational notifications like *Get Admin Key* and *Get Query keys*. These activities indicate requests that were made using the admin key (create or delete objects) or query key, but do not show the request itself. For information of this grain, you must configure diagnostic logging.

You can access the [Activity log](#) from the left-navigation pane, or from Notifications in the top window command bar, or from the [Diagnose and solve problems](#) page.

Monitor storage in the Usage tab

For visual monitoring in the portal, the [Usage](#) tab shows you resource availability relative to current [limits](#) imposed by the service tier. If you are finalizing decisions about [which tier to use for production workloads](#), or whether to [adjust the number of active replicas and partitions](#), these metrics can help you with those decisions by showing you how quickly resources are consumed and how well the current configuration handles the existing load.

The following illustration is for the free service, which is capped at 3 objects of each type and 50 MB of storage. A Basic or Standard service has higher limits, and if you increase the partition counts, maximum storage goes up proportionally.



NOTE

Alerts related to storage are not currently available; storage consumption is not aggregated or logged into the **AzureMetrics** table in Azure Monitor. To get storage alerts, you would need to [build a custom solution](#) that emits resource-related notifications, where your code checks for storage size and handles the response.

Add-on monitoring with Azure Monitor

Many services, including Azure Cognitive Search, integrate with [Azure Monitor](#) for additional alerts, metrics, and logging diagnostic data.

[Enable diagnostic logging](#) for a search service if you want control over data collection and storage. Logged events captured by Azure Monitor are stored in the **AzureDiagnostics** table and consists of operational data related to queries and indexing.

Azure Monitor provides several storage options, and your choice determines how you can consume the data:

- Choose Azure Blob storage if you want to [visualize log data](#) in a Power BI report.
- Choose Log Analytics if you want to explore data through Kusto queries.

Azure Monitor has its own billing structure and the diagnostic logs referenced in this section have an associated cost. For more information, see [Usage and estimated costs in Azure Monitor](#).

Monitor user access

Because search indexes are a component of a larger client application, there is no built-in methodology for controlling or monitoring per-user access to an index. Requests are assumed to come from a client application, for either admin or query requests. Admin read-write operations include creating, updating, deleting objects across the entire service. Read-only operations are queries against the documents collection, scoped to a single index.

As such, what you'll see in the activity logs are references to calls using admin keys or query keys. The appropriate key is included in requests originating from client code. The service is not equipped to handle identity tokens or impersonation.

When business requirements do exist for per-user authorization, the recommendation is integration with Azure Active Directory. You can use \$filter and user identities to [trim search results](#) of documents that a user should not

see.

There is no way to log this information separately from the query string that includes the \$filter parameter. See [Monitor queries](#) for details on reporting query strings.

Next steps

Fluency with Azure Monitor is essential for oversight of any Azure service, including resources like Azure Cognitive Search. If you are not familiar with Azure Monitor, take the time to review articles related to resources. In addition to tutorials, the following article is a good place to start.

[Monitoring Azure resources with Azure Monitor](#)

Collect and analyze log data for Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

Diagnostic or operational logs provide insight into the detailed operations of Azure Cognitive Search and are useful for monitoring service and workload processes. Internally, some system information exists on the backend for a short period of time, sufficient for investigation and analysis if you file a support ticket. However, if you want self-direction over operational data, you should configure a diagnostic setting to specify where logging information is collected.

Diagnostic logging is enabled through integration with [Azure Monitor](#).

When you set up diagnostic logging, you will be asked to specify a storage mechanism. The following table enumerates options for collecting and persisting data.

RESOURCE	USED FOR
Send to Log Analytics workspace	Events and metrics are sent to a Log Analytics workspace, which can be queried in the portal to return detailed information. For an introduction, see Get started with Azure Monitor logs
Archive with Blob storage	Events and metrics are archived to a Blob container and stored in JSON files. Logs can be quite granular (by the hour/minute), useful for researching a specific incident but not for open-ended investigation. Use a JSON editor to view a raw log file or Power BI to aggregate and visualize log data.
Stream to Event Hub	Events and metrics are streamed to an Azure Event Hubs service. Choose this as an alternative data collection service for very large logs.

Prerequisites

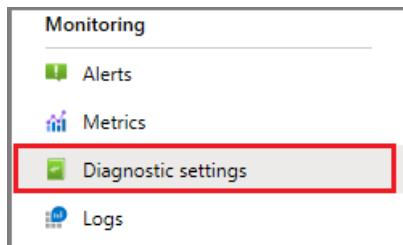
Create resources in advance so that you can select one or more when configuring diagnostic logging.

- [Create a log analytics workspace](#)
- [Create a storage account](#)
- [Create an Event Hub](#)

Enable data collection

Diagnostic settings specify how logged events and metrics are collected.

1. Under **Monitoring**, select **Diagnostic settings**.



2. Select + Add diagnostic setting

3. Check Log Analytics, select your workspace, and select OperationLogs and AllMetrics.

A screenshot of the 'Diagnostics settings' configuration page. It shows the following fields:

- Name: demo-diagnostics
- Archive to a storage account: Unchecked
- Stream to an event hub: Unchecked
- Send to Log Analytics: Checked
- Subscription: demo-subscription
- Log Analytics workspace: demo-westus2 (westus2)
- log: OperationLogs (checked)
- metric: AllMetrics (checked)

The 'Send to Log Analytics' section and its sub-sections are highlighted with a red box.

4. Save the setting.

5. After logging has been enabled, use your search service to start generating logs and metrics. It will take time before logged events and metrics become available.

For Log Analytics, it will be several minutes before data is available, after which you can run Kusto queries to return data. For more information, see [Monitor query requests](#).

For Blob storage, it takes one hour before the containers will appear in Blob storage. There is one blob, per hour, per container. Containers are only created when there is an activity to log or measure. When the data is copied to a storage account, the data is formatted as JSON and placed in two containers:

- insights-logs-operationlogs: for search traffic logs
- insights-metrics-pt1m: for metrics

Query log information

Two tables contain logs and metrics for Azure Cognitive Search: **AzureDiagnostics** and **AzureMetrics**.

- Under **Monitoring**, select **Logs**.
- Enter **AzureMetrics** in the query window. Run this simple query to get acquainted with the data collected in this table. Scroll across the table to view metrics and values. Notice the record count at the top, and if your service has been collecting metrics for a while, you might want to adjust the time interval to get a manageable data set.

AzureMetrics

Completed. Showing partial results from the last 24 hours. 00:00:00.984 21 records

Table

MetricName	Total	Count	Maximum	Minimum	Average	TimeGrain
ThrottledSearchQueriesPercentage	0	1	0	0	0	PT1M
SearchQueriesPerSecond	4	4	1	1	1	PT1M
SearchLatency	0	6	0	0	0	PT1M
SearchQueriesPerSecond	6	6	1	1	1	PT1M
SearchLatency	0	3	0	0	0	PT1M

- Enter the following query to return a tabular result set.

```
AzureMetrics
| project MetricName, Total, Count, Maximum, Minimum, Average
```

- Repeat the previous steps, starting with **AzureDiagnostics** to return all columns for informational purposes, followed by a more selective query that extracts more interesting information.

```
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d, IndexName_s
| where OperationName == "Query.Search"
```

AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d, IndexName_s
| where OperationName == "Query.Search"

Completed. Showing partial results from the last 24 hours.

00:00:01.386 23 records

Table Chart Columns ↕ Copy request ID

Drag a column header and drop it here to group by that column

Operation...	resu...	Du...	Query_s	Doc...	IndexName_s
› Query.Search	200	8	?api-version=2019-05-06&search=seattle%20hotels	4	hotels-quicks...
› Query.Search	200	107	?api-version=2019-05-06&search=seattle%20hotels&%24count=true	4	hotels-quicks...
› Query.Search	200	68	?api-version=2019-05-06&search=washington&%24count=true	0	hotels-quicks...
› Query.Search	200	148	?api-version=2019-05-06&search=*&%24count=true&%24select=Hot...	4	hotels-quicks...
› Query.Search	200	636	?api-version=2019-05-06&search=restaurant&%24count=true&%24se...	1	hotels-quicks...
› Query.Search	200	128	?api-version=2019-05-06&search=*	14	demoindex
› Query.Search	200	138	?api-version=2017-11-11&search=*&%24count=true	14	demoindex
› Query.Search	200	118	?api-version=2019-05-06&search=*	19	hotel-reviews

Kusto query examples

If you enabled diagnostic logging, you can query **AzureDiagnostics** for a list of operations that ran on your service and when. You can also correlate activity to investigate changes in performance.

Example: List operations

Return a list of operations and a count of each one.

```
AzureDiagnostics  
| summarize count() by OperationName
```

Example: Correlate operations

Correlate query request with indexing operations, and render the data points across a time chart to see operations coincide.

```
AzureDiagnostics  
| summarize OperationName, Count=count()  
| where OperationName in ('Query.Search', 'Indexing.Index')  
| summarize Count=count(), AvgLatency=avg(DurationMs) by bin(TimeGenerated, 1h), OperationName  
| render timechart
```

Logged operations

Logged events captured by Azure Monitor include those related to indexing and queries. The **AzureDiagnostics** table in Log Analytics collects operational data related to queries and indexing.

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.

OPERATIONNAME	DESCRIPTION
Query.Search	Query requests against an index. See Monitor queries for information about logged queries.
Indexing.Index	This operation is a call to Add, Update or Delete Documents .
indexes.Prototype	This is an index created by the Import Data wizard.
Indexers.Create	Create an indexer explicitly or implicitly through the Import Data wizard.
Indexers.Get	Returns the name of an indexer whenever the indexer is run.
Indexers.Status	Returns the status of an indexer whenever the indexer is run.
DataSources.Get	Returns the name of the data source whenever an indexer is run.
Indexes.Get	Returns the name of an index whenever an indexer is run.

Log schema

If you are building custom reports, the data structures that contain Azure Cognitive Search log data conform to the schema below. For 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.

The following table is a partial list of fields common to resource logging.

NAME	TYPE	EXAMPLE	NOTES
timeGenerated	datetime	"2018-12-07T00:00:43.6872559Z"	Timestamp of the operation
resourceId	string	"/SUBSCRIPTIONS/11111111-1111-1111-1111-111111111111/RESOURCEGROUPS/DEFAUL/PROVIDERS/MICROSOFTSEARCH/SEARCHSERVICES/SEARCHSERVICE"	Your ResourceId
operationName	string	"Query.Search"	The name of the operation
operationVersion	string	"2020-06-30"	The api-version used
category	string	"OperationLogs"	constant
resultType	string	"Success"	Possible values: Success or Failure
resultSignature	int	200	HTTP result code

NAME	TYPE	EXAMPLE	NOTES
durationMS	int	50	Duration of the operation in milliseconds
properties	object	see the following table	Object containing operation-specific data

Properties schema

The properties below are specific to Azure Cognitive Search.

NAME	TYPE	EXAMPLE	NOTES
Description_s	string	"GET /indexes('content')/docs"	The operation's endpoint
Documents_d	int	42	Number of documents processed
IndexName_s	string	"test-index"	Name of the index associated with the operation
Query_s	string	"?search=AzureSearch&\$count=true&api-version=2020-06-30"	The query parameters

Metrics schema

Metrics are captured for query requests and measured in one minute intervals. Every metric exposes minimum, maximum and average values per minute. For more information, see [Monitor query requests](#).

NAME	TYPE	EXAMPLE	NOTES
resourceId	string	"/SUBSCRIPTIONS/11111111-1111-1111-1111-111111111111/RESOURCEGROUPS/DEFAULPROVIDERS/MICROSOFTSEARCH/SEARCHSERVICES/SEARCHSERVICE"	your resource ID
metricName	string	"Latency"	the name of the metric
time	datetime	"2018-12-07T00:00:43.6872559Z"	the operation's timestamp
average	int	64	The average value of the raw samples in the metric time interval, units in seconds or percentage, depending on the metric.

NAME	TYPE	EXAMPLE	NOTES
minimum	int	37	The minimum value of the raw samples in the metric time interval, units in seconds.
maximum	int	78	The maximum value of the raw samples in the metric time interval, units in seconds.
total	int	258	The total value of the raw samples in the metric time interval, units in seconds.
count	int	4	The number of metrics emitted from a node to the log within the one minute interval.
timegrain	string	"PT1M"	The time grain of the metric in ISO 8601.

It's common for queries to execute in milliseconds, so only queries that measure as seconds will appear in metric like QPS.

For the **Search Queries Per Second** metric, 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.

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.

View raw log files

Blob storage is used for archiving log files. You can use any JSON editor to view the log file. If you don't have one, we recommend [Visual Studio Code](#).

1. In Azure portal, open your Storage account.
2. In the left-navigation pane, click **Blobs**. You should see **insights-logs-operationlogs** and **insights-metrics-pt1m**. These containers are created by Azure Cognitive Search when the log data is exported to Blob storage.
3. Click down the folder hierarchy until you reach the .json file. Use the context-menu to download the file.

Once the file is downloaded, open it in a JSON editor to view the contents.

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](#)

Visualize Azure Cognitive Search Logs and Metrics with Power BI

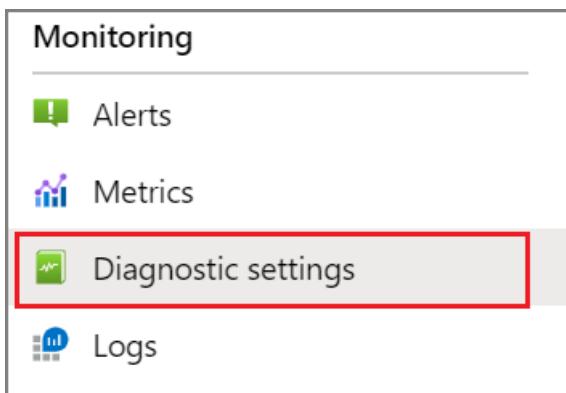
10/4/2020 • 3 minutes to read • [Edit Online](#)

Azure Cognitive Search allows you to store operation logs and service metrics about your search service in an Azure Storage account. This page provides instructions for how you can visualize that information through a Power BI Template App. The app provides detailed insights about your search service, including information about Search, 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](#).

How to get started with 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**

Name	demo-diagnostics-settings
<input checked="" type="checkbox"/> Archive to a storage account	
Storage account >	
logsandmetricsstorage	
<input type="checkbox"/> Stream to an event hub	
<input type="checkbox"/> Send to Log Analytics	
log	
<input checked="" type="checkbox"/> OperationLogs	Retention (days) ⓘ <input type="range" value="0"/> 0
metric	
<input checked="" type="checkbox"/> AllMetrics	Retention (days) ⓘ <input type="range" value="0"/> 0

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

Get started with your new app

Explore your app with sample data, go to the workspace to customize as needed and share with your organization, or connect your data to get up and running.

Connect your data

Connect to a data source to view your new app with your own data.



Connect

Explore with sample data

Open your new app to start exploring with sample data.



Explore app

Customize and share

Your app comes with a workspace, so you can customize and share it, just like an app you built yourself.



Edit workspace

[Don't show this again](#)

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.

Connect to Azure Cognitive Search

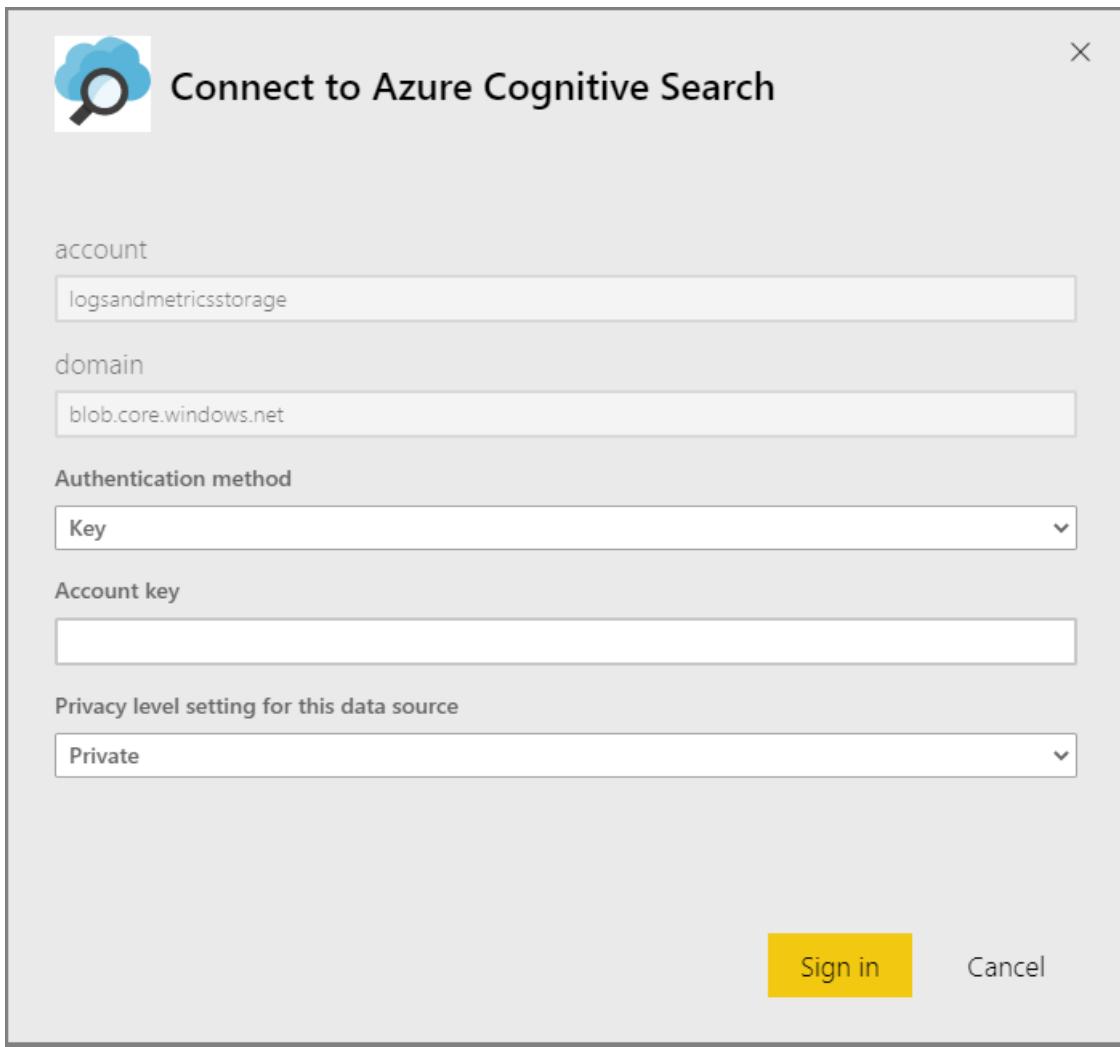
Before connecting to your data, you must update the required parameters (*).

StorageAccount
The Azure Storage account name you use for Azure Cognitive Search Logs and Metrics. More information on how to enable logging can be found here: <https://docs.microsoft.com/azure/search/search-monitor-usage>

Days
Number of days of data to query.

Next **Cancel**

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.



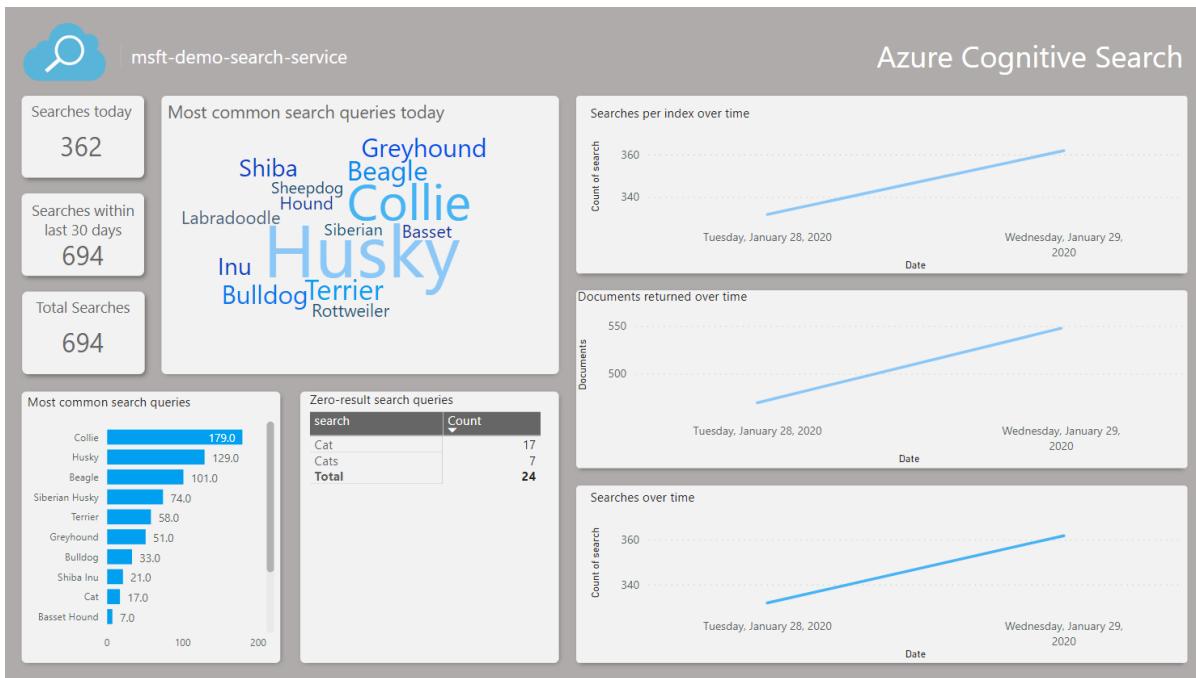
8. Wait for the data to refresh. This may take some time depending on how much data you have. You can see if the data is still being refreshed based on the below indicator.

Name	Type	Sensitivity (preview)	Owner	Refreshed	Endorsement	Include in app
Azure Cognitive Search Dataset	Dataset	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	
Azure Cognitive Search Report	Report	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	Yes

9. Once the data refresh has completed, select **Azure Cognitive Search Report** to view the report.

Name	Type	Sensitivity (preview)	Owner	Refreshed	Endorsement	Include in app
Azure Cognitive Search Dataset	Dataset	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	
Azure Cognitive Search Report	Report	—	Azure Cognitive Sear...	2/7/20, 4:01:01 PM	—	Yes

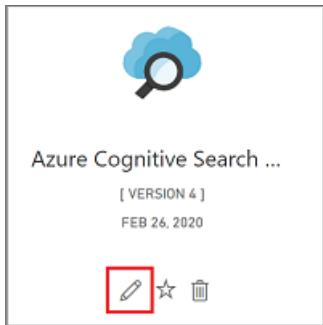
10. Make sure to refresh the page after opening the report so that it opens with your data.



How to change the 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.

Name	Type	Sensitivity (preview)
 Azure Cognitive Search Dataset	Dataset	—
 Azure Cognitive Search Report		<ul style="list-style-type: none"> Analyze in Excel Create report Delete Get quick insights Refresh now Rename Schedule refresh Settings Download .pbix Manage permissions View related

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.

Name	Type	Sensitivity (preview)
 Azure Cognitive Search Dataset	Dataset	—
 Azure Cognitive Search Report		<ul style="list-style-type: none"> Analyze in Excel Create report Delete Get quick insights Refresh now Rename Schedule refresh Settings Download .pbix Manage permissions View related

5. Open the report to view the updated data. You might also need to refresh the report to view the latest data.

Troubleshooting

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

[Learn more about Azure Cognitive Search](#)

[What is Power BI?](#)

[Basic concepts for designers in the Power BI service](#)

Monitor query requests in Azure Cognitive Search

10/4/2020 • 8 minutes to read • [Edit Online](#)

This article explains how to measure query performance and volume using metrics and resource logging. It also explains how to collect the input terms used in queries - necessary information when you need to assess the utility and effectiveness of your search corpus.

Historical data that feeds into metrics is preserved for 30 days. For longer retention, or to report on operational data and query strings, be sure to enable a [diagnostic setting](#) that specifies a storage option for persisting logged events 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:

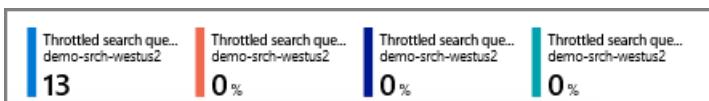
- A service returns an error "You are sending too many requests. Please try again later."
- A service returns a 503 error code indicating the service is currently unavailable.
- If you are using the portal (for example, Search Explorer), the query is dropped silently and you will need to click Search 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.
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.

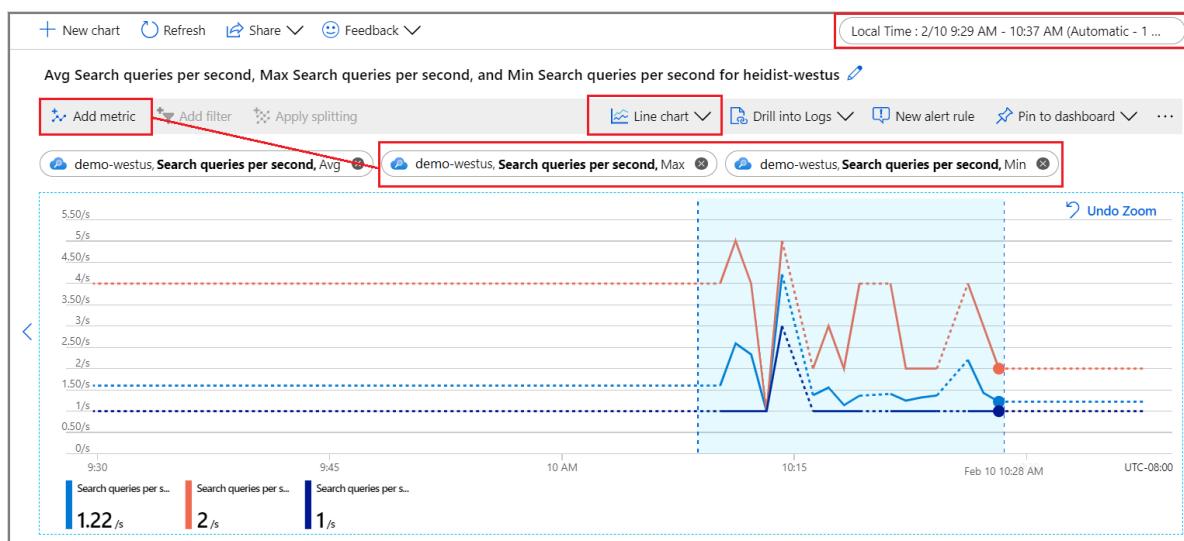


Explore metrics in the portal

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](#).

- Under the Monitoring section, select **Metrics** to open the metrics explorer with the scope set to your search service.
- 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.



- 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.

Identify strings used in queries

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.

```
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`.

The screenshot shows the Azure Log Analytics query editor interface. At the top, there's a query bar with the following code:

```
AzureDiagnostics
| project OperationName, Query_s, IndexName_s, Documents_d
| where OperationName == "Query.Search"
| where Query_s != "?api-version=2019-05-06&search=*"
| where IndexName_s != "realestate-us-sample-index"
```

The results pane below shows a table with columns: OperationName, Query_s, IndexName_s, and Documents_d. The data is grouped by OperationName and Query_s. A filter sidebar on the right is expanded, showing a dropdown menu for 'IndexName_s' with the option 'Is equal to' selected. Below it, there are two more dropdown menus for 'Documents_d' and a 'Filter' button.

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 query to return queries, sorted by duration in milliseconds. The longest-running queries are at the top.

```
AzureDiagnostics
| project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d, IndexName_s
| where OperationName == "Query.Search"
| sort by DurationMs
```

The screenshot shows the Azure Metrics Explorer interface. At the top, there are buttons for 'Run' (highlighted in blue), 'Time range : Last 24 hours', 'Save', 'Copy link', 'New alert rule', 'Export', 'Pin to dashboard', and 'Prettify query'. Below the toolbar, the search query is displayed: 'AzureDiagnostics | project OperationName, resultSignature_d, DurationMs, Query_s, Documents_d, IndexName_s | where OperationName == "Query.Search" | sort by DurationMs'. The results section starts with the message 'Completed. Showing partial results from the last 24 hours.' It shows 23 records with a total duration of 00:00:04.556. The results table has columns: OperationName, DurationMs, Query_s, Do..., and IndexName_s. The first six rows are highlighted with a red box.

OperationName	DurationMs	Query_s	Do...	IndexName_s
Query.Search	200	?api-version=2019-05-06&search=restaurant&%24count=true&%24select=Hotell...	1	hotels-quickstart
Query.Search	200	?api-version=2019-05-06&search=*&%24count=true	14	demoindex
Query.Search	200	?api-version=2019-05-06&search=*	14	demoindex
Query.Search	200	?api-version=2019-05-06-Preview&search=*&%24count=true	14	demoindex
Query.Search	200	?api-version=2019-05-06&search=*&%24count=true&%24select=HotelId%2C%	4	hotels-quickstart
Query.Search	200	?api-version=2019-05-06&search=*	19	hotel-reviews-idx

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.

- Under the Monitoring section, select **Alerts** and then click **+ New alert rule**. Make sure your search service is selected as the resource.
- Under Condition, click Add.
- Configure signal logic. For signal type, choose **metrics** and then select the signal.
- 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.
- Next, scroll down to Alert logic. For proof-of-concept, you could specify an artificially low value for testing purposes.

Alert logic

Threshold ⓘ

Static **Dynamic**

Operator ⓘ **Greater than** **Aggregation type** * ⓘ **Average** **Threshold value** * ⓘ **0** **count/second**

Condition preview

Whenever the search queries per second is greaterthan 0 count/second

Evaluated based on

Aggregation granularity (Period) * ⓘ **5 minutes** **Frequency of evaluation** ⓘ **Every 1 Minute**

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.

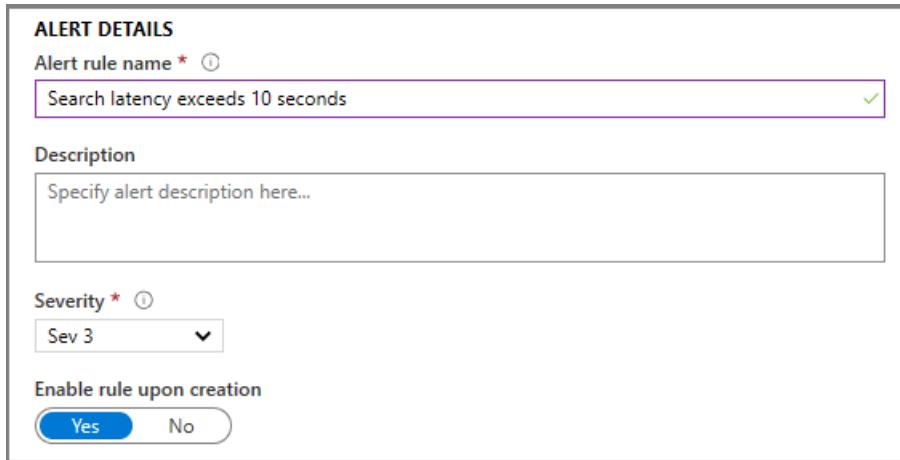
ALERT DETAILS

Alert rule name * ⓘ
Search latency exceeds 10 seconds ✓

Description
Specify alert description here...

Severity * ⓘ
Sev 3

Enable rule upon creation



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](#)

Collect telemetry data for search traffic analytics

10/4/2020 • 8 minutes to read • [Edit Online](#)

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. Search requests used to populate facets, additional content or any internal information, are not important and they skew and bias 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, the Search Traffic Analytics page contains 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.

The screenshot shows the Azure portal's 'Search traffic analytics' blade. On the left, a sidebar lists various monitoring and support options. The main area has three main sections:

- 1. Select the Application Insights resource you will use:** A radio button for 'Use existing' is selected. A dropdown menu labeled 'Select your resource' is open. Below it are fields for 'Resource name', 'Subscription ID', 'Instrumentation Key', and 'Resource group'.
- 2. Copy these snippets into your application code to add search telemetry:** A tab for 'JavaScript' is selected. Under 'Telemetry client', the 'Headers' section contains the following code:


```
1 request.setRequestHeader("x-ms-azs-return-searchid", "true");
2 request.setRequestHeader("Access-Control-Expose-Headers", "x-ms-azs-searchid");
3 var searchId = request.getResponseHeader('x-ms-azs-searchid');
```
- 3. Monitor your search metrics with Power BI:** A note says 'On download, use the Application Insights resource information on this page to connect your report to analytical data captured for your application.' It includes links to 'Download PowerBI report' and 'Get PowerBI desktop'.

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 you 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** is found 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.

```
private static TelemetryClient _telemetryClient;

// Add a constructor that accepts a telemetry client:
public HomeController(TelemetryClient telemetry)
{
    _telemetryClient = telemetry;
}
```

Use JavaScript

```
<script type="text/javascript">var appInsights=window.appInsights||function(config){function r(config)
{t[config]=function(){var i=arguments;t.queue.push(function(){t[config].apply(t,i)})}}var t=
{config:config},u=document,e=window,o="script",s=u.createElement(o),i,f;s.src=config.url||"/az416426.vo.msecdn.net/scripts/a/ai.0.js";u.getElementsByTagName(o)[0].parentNode.appendChild(s);try{t.cookie=u.cookie}catch(h)
{}for(t.queue=[],i=
["Event","Exception","Metric","PageView","Trace","Dependency"];i.length;)r("track"+i.pop());return
r("setAuthenticatedUserContext"),r("clearAuthenticatedUserContext"),config.disableExceptionTracking||
(i=="onerror",r("_"+i),f=e[i],e[i]=function(config,r,u,e,o){var s=f&&f(config,r,u,e,o);return s!==!0&&t["_"+i]
(config,r,u,e,o),s}),t
){}
instrumentationKey: "<YOUR INSTRUMENTATION KEY>"
});
window.appInsights=appInsights;
</script>
```

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#

```
// 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)

```
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#

```
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

```
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#

```
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

```
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. Monitor your search metrics with Power BI

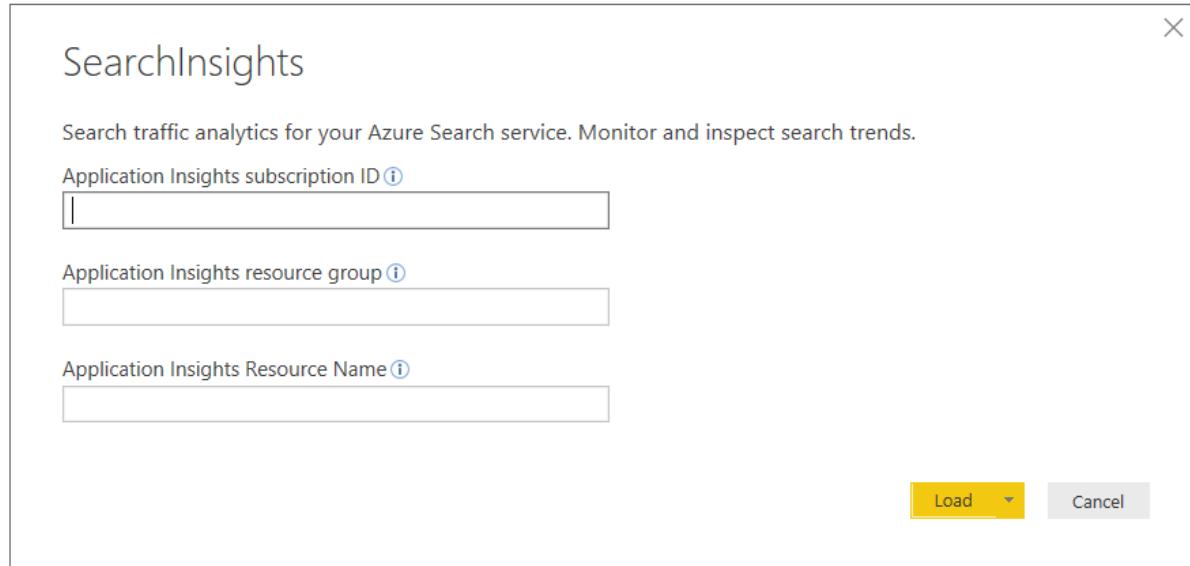
On download, use the Application Insights resource information on this page to connect your report to analytical data captured for your application.

[Download PowerBI report](#)

[Get PowerBI desktop](#)

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**.

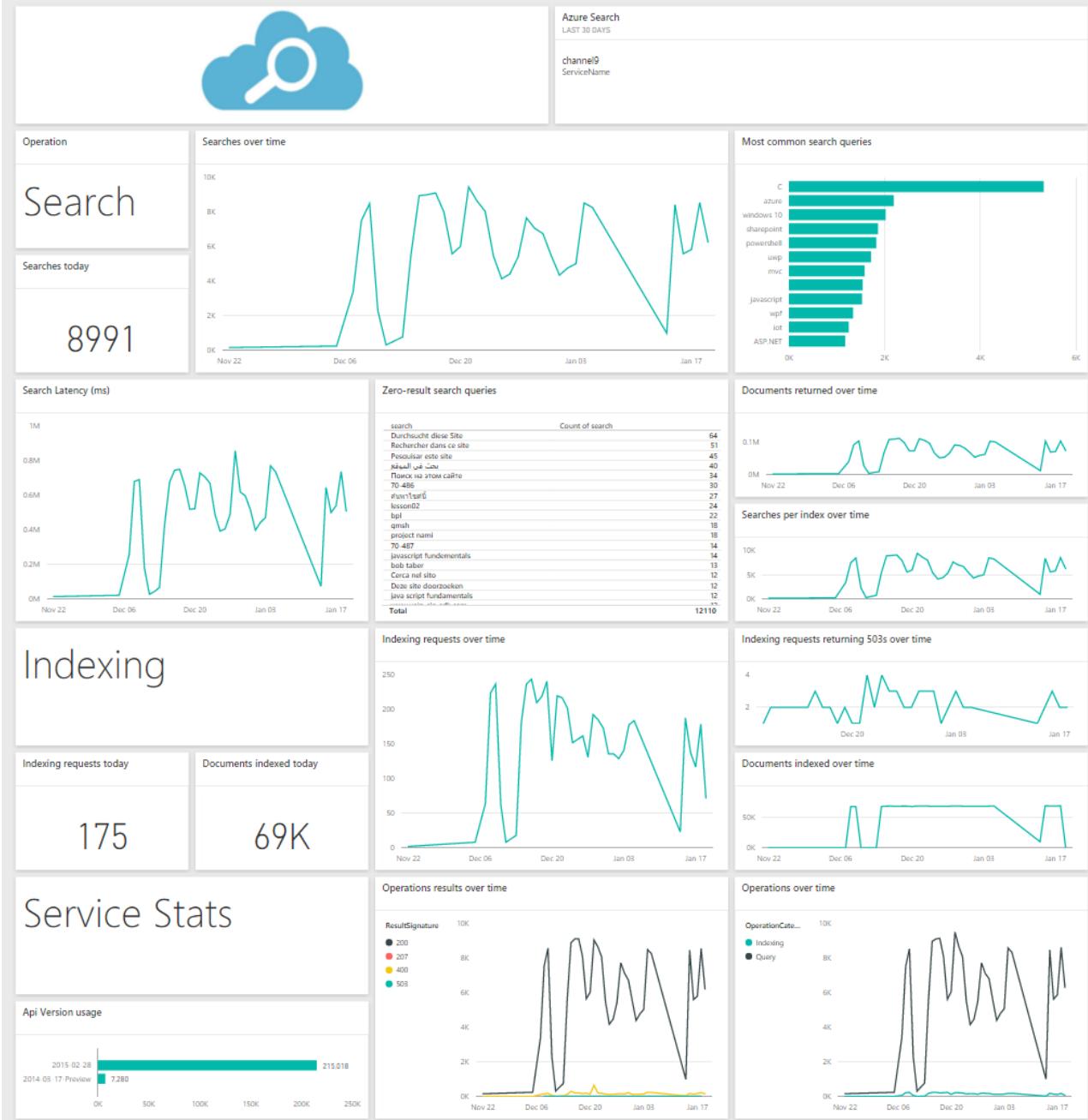
The report contains charts and tables that help you make more informed decisions to improve your search performance and relevance.

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.

AZURE SEARCH



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.

Troubleshooting common indexer issues in Azure Cognitive Search

10/4/2020 • 4 minutes to read • [Edit Online](#)

Indexers can run into a number of issues when indexing data into Azure Cognitive Search. The main categories of failure include:

- [Connecting to a data source or other resources](#)
- [Document processing](#)
- [Document ingestion to an index](#)

Connection errors

NOTE

Indexers have limited support for accessing data sources and other resources that are secured by Azure network security mechanisms. Currently, indexers can only access data sources via corresponding IP address range restriction mechanisms or NSG rules when applicable. Details for accessing each supported data source can be found below.

You can find out the IP address of your search service by pinging its fully qualified domain name (eg.,

`<your-search-service-name>.search.windows.net`).

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.

Configure firewall rules

Azure Storage, CosmosDB and Azure SQL provide a configurable firewall. There's no specific error message when the firewall is enabled. Typically, firewall errors are generic and look like

`The remote server returned an error: (403) Forbidden` or

`Credentials provided in the connection string are invalid or have expired`.

There are 2 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](#)
- [Cosmos DB](#)
- [Azure SQL](#)

Limitation: As stated in the documentation above for Azure Storage, IP address range restrictions will 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](#).

Details for accessing data in SQL server on an Azure VM are outlined [here](#)

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](#)

CosmosDB "Indexing" isn't enabled

Azure Cognitive Search has an implicit dependency on Cosmos DB indexing. If you turn off automatic indexing in 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](#).

Document processing errors

Unprocessable or unsupported documents

The blob indexer [documents which document formats are explicitly supported](#). Sometimes, a blob storage container contains unsupported documents. Other times there may be problematic documents. You can avoid stopping your indexer on these documents by [changing configuration options](#):

```
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 document content

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 [cognitive search](#) 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](#):

```
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" } }
}
```

Index errors

Missing documents

Indexers find documents from a [data source](#). Sometimes a document from the data source that should have been indexed appears to be missing from an index. There are a couple of common reasons these errors may happen:

- The document hasn't been indexed. Check the portal for a successful indexer run.
- Check your [change tracking](#) value. If your high watermark value is a date set to a future time, then any documents that have a date less than this will be skipped by the indexer. You can understand your indexer's change tracking state using the 'initialTrackingState' and 'finalTrackingState' fields in the [indexer status](#).
- The document was updated after the indexer run. If your indexer is on a [schedule](#), it will eventually rerun and pick up the document.
- The [query](#) specified in the data source excludes the document. Indexers can't index documents that aren't part of the data source.
- [Field mappings](#) or [AI enrichment](#) have changed the document and it looks different than you expect.
- Use the [lookup document API](#) to find your document.

Troubleshooting common indexer errors and warnings in Azure Cognitive Search

10/4/2020 • 23 minutes to read • [Edit Online](#)

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 do not 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://coromsearch.blob.core.windows.net/jf-k-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.	<pre>link-cryptonyms-list - Error processing the request record : System.ArgumentNullException: Value cannot be null. Parameter name: source at System.Linq.Enumerable.All[TSource](IEnumerable<TSource> source, Func<TSource, bool> predicate) at Microsoft.CognitiveSearch.WebApiSkills.JfkWebAp...rest of stack trace...</pre>
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
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 Cosmos DB) <code>{"Errors": ["Request rate is large"]}</code>	Check your storage instance to ensure it's healthy. You may need to adjust your scaling/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.

REASON	DETAILS/EXAMPLE	RESOLUTION
Could not apply field mapping to a field	Could not apply mapping function <code>'functionName'</code> to field <code>'fieldName'</code> . 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.
Could not read field value	Could not read the value of column <code>'fieldName'</code> at index <code>'fieldIndex'</code> . 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 are 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

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 mean there is a product bug. Please 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

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 will 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

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 will need to debug your custom code to resolve the issue. If instead the failure is from a built-in skill, please file a [support ticket](#) to get assistance.

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. Please 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, please file a [support ticket](#) to get assistance, as this is not expected.

Custom skills

If you encounter a timeout error with a custom skill you have created, there are a couple of things you can try. First, review your custom skill and ensure that it is not getting stuck in an infinite loop and that it is returning a result consistently. Once you have confirmed that is the case, determine what the execution time of your skill is. If you didn't explicitly set a `timeout` value on your custom skill definition, then the default `timeout` is 30 seconds. If 30 seconds is not long enough for your skill to execute, you may specify a higher `timeout` value on your custom skill definition. Here is an example of a custom skill definition where the timeout is set to 90 seconds:

```
{
    "@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 will 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 is not 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

Reason	Details/Example	Resolution
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. Please 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.'
An unknown type was discovered in the source document.	Unknown type ' <i>unknown</i> ' cannot be indexed
An incompatible notation for geography points was used in the source document.	WKT POINT string literals are not supported. Please 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 does not 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 cannot afford to skip any document, or if you are 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 in our attempt to do so. 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 are a set of known failure states and possible resolutions.

REASON	DETAILS/EXAMPLE	RESOLUTION
Could not update projection blob <code>'blobUri'</code> in container <code>'containerName'</code>	The specified container does not 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 <code>'blobUri'</code> in container <code>'containerName'</code>	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, please file a support ticket so it can be investigated further.
Could not update row <code>'projectionRow'</code> in table <code>'tableName'</code>	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, please file a support ticket so it can be investigated further.

Warning: Skill input was invalid

An input to the skill was missing, the wrong type, or otherwise invalid. The warning message will indicate the impact:

1. Could not execute skill
2. 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 any required inputs are missing or if any input is not the right type, the skill gets skipped and generates a warning. Skipped skills do not generate any outputs, so if other skills use outputs of the skipped skill they may generate additional warnings.

If an optional input is missing, the skill will still run 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 do not have an entry for `middleName`. If you to pass `middleName` as an input to a skill in the pipeline, then it is 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.

```
{  
    "@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 <code>String</code> . Name: <code>text</code> , Source: <code>/document/merged_content</code> ." "Required skill input was not of the expected format. Name: <code>text</code> , Source: <code>/document/merged_content</code> ." "Cannot iterate over non-array <code>/document/normalized_images/0/imageCelebrities</code> ." "Unable to select <code>0</code> in non-array <code>/document/normalized_images/0/imageCelebrities/0/detail/celebrities</code> ." " 	Certain skills expect inputs of particular types, for example Sentiment skill expects <code>text</code> 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 <code>*</code> in the correct positions. Usually both the context and input source should end with <code>*</code> for arrays.
Skill input is missing	"Required skill input is missing. Name: <code>text</code> , Source: <code>/document/merged_content</code> ." "Missing value <code>/document/normalized_images/0/imageTags</code> ." "Unable to select <code>0</code> in array <code>/document/pages</code> of length <code>0</code> ." 	If all your documents get this warning, most likely there is a typo in the input paths and you should double check property name casing, extra or missing <code>*</code> in the path, and make sure that the documents from the data source provide the required inputs.
Skill language code input is invalid	Skill input <code>languageCode</code> has the following language codes <code>X,Y,Z</code> , 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 is not supported. This can occur if you are passing the output of the [LanguageDetectionSkill](#) to subsequent skills, and the output consists of more languages than are supported in those downstream skills.

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 is an example of what this might look like for the EntityRecognitionSkill:

```
{
  "@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:

- [Text Analytics Supported Languages](#) (for the [KeyPhraseExtractionSkill](#), [EntityRecognitionSkill](#), [SentimentSkill](#), and [PIIDetectionSkill](#))
- [Translator Supported Languages](#) (for the [Text TranslationSkill](#))
- [Text SplitSkill](#) Supported Languages: `da, de, en, es, fi, fr, it, ko, pt`

Warning: Skill input was truncated

Cognitive skills have limits to the length of text that can be analyzed at once. If the text input of these skills are over that limit, we will truncate the text to meet the limit, and then perform the enrichment on that truncated text. This means that the skill is executed, but not over all of your data.

In the example LanguageDetectionSkill below, the `'text'` input field may trigger this warning if it is over the character limit. You can find the skill input limits in the [skills documentation](#).

```
{
  "@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

Indexer was able to run a skill in the skillset, but the response from the Web API request indicated there were warnings during execution. Review the warnings to understand how your data is impacted and whether or not action is required.

Warning: The current indexer configuration does not support incremental progress

This warning only occurs for 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 is 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 will not 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 workaround 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 <code>/document/normalized_images/0/imageCelebrity</code> array field 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 <code>*</code> 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.	This error occurs when the output is not an array, check the indicated output source field path for errors. For example, you might have a missing or extra <code>*</code> 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 <code>0</code> in non-array	"Unable to select <code>0</code> in non-array <code>/document/pages</code> ." This could happen if the skills output does not produce an array and the output source field name has array index or <code>*</code> in its path. Please 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.	This could happen if the skills output does not produce an array and the output source field name has array index or <code>*</code> in its path. Please 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

ENCODING	BYTE ORDER MARK
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: 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 Text Analytics service. In some cases, it is 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.

OData language overview for `$filter`, `$orderby`, and `$select` in Azure Cognitive Search

10/4/2020 • 8 minutes to read • [Edit Online](#)

Azure Cognitive Search supports a subset of the OData expression syntax for `$filter`, `$orderby`, and `$select` expressions. Filter expressions are evaluated during query parsing, constraining search to specific fields or adding match criteria used during index scans. Order-by 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 that is used in the `search` parameter, although there's some overlap in the syntax for referencing fields.

This article provides an overview of the OData expression language used in filters, order-by, and select expressions. The language is presented "bottom-up", starting with the most basic elements and building on them. The top-level syntax for each parameter is described in a separate article:

- [\\$filter syntax](#)
- [\\$orderby syntax](#)
- [\\$select syntax](#)

OData expressions range from simple to highly complex, but they all share common elements. The most basic parts of an OData expression in Azure Cognitive Search are:

- **Field paths**, which refer to specific fields of your index.
- **Constants**, which are literal values of a certain data type.

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 `a11`) 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 sub-fields 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
<code>HotelName</code>	Refers to a top-level field of the index
<code>Address/City</code>	Refers to the <code>city</code> sub-field of a complex field in the index; <code>Address</code> is of type <code>Edm.ComplexType</code> in this example
<code>Rooms/Type</code>	Refers to the <code>Type</code> sub-field of a complex collection field in the index; <code>Rooms</code> is of type <code>Collection(Edm.ComplexType)</code> in this example
<code>Stores/Address/Country</code>	Refers to the <code>Country</code> sub-field of the <code>Address</code> sub-field of a complex collection field in the index; <code>Stores</code> is of type <code>Collection(Edm.ComplexType)</code> and <code>Address</code> is of type <code>Edm.ComplexType</code> in this example
<code>room/Type</code>	Refers to the <code>Type</code> sub-field of the <code>room</code> range variable, for example in the filter expression <code>Rooms/any(room: room/Type eq 'deluxe')</code>
<code>store/Address/Country</code>	Refers to the <code>Country</code> sub-field of the <code>Address</code> sub-field of the <code>store</code> range variable, for example in the filter expression <code>Stores/any(store: store/Address/Country eq 'Canada')</code>

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` sub-field 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 sub-field `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:

```
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` sub-field, 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	<code>suggesters/sourceFields</code>	None
Create or Update Index	<code>scoringProfiles/text/weights</code>	Can only refer to searchable fields
Create or Update Index	<code>scoringProfiles/functions/fieldName</code>	Can only refer to filterable fields
Search	<code>search</code> when <code>queryType</code> is <code>full</code>	Can only refer to searchable fields
Search	<code>facet</code>	Can only refer to facetable fields
Search	<code>highlight</code>	Can only refer to searchable fields
Search	<code>searchFields</code>	Can only refer to searchable fields
Suggest and Autocomplete	<code>searchFields</code>	Can only refer to fields that are part of a suggester
Search, Suggest, and Autocomplete	<code>\$filter</code>	Can only refer to filterable fields
Search and Suggest	<code>\$orderby</code>	Can only refer to sortable fields
Search, Suggest, and Lookup	<code>\$select</code>	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
<code>Edm.Boolean</code>	<code>true</code> , <code>false</code>
<code>Edm.DateTimeOffset</code>	<code>2019-05-06T12:30:05.451Z</code>
<code>Edm.Double</code>	<code>3.14159</code> , <code>-1.2e7</code> , <code>NaN</code> , <code>INF</code> , <code>-INF</code>

DATA TYPE	EXAMPLE CONSTANTS
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))'
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.

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 sub-expressions. These sub-expressions 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](#)

See also

- [Faceted navigation in Azure Cognitive Search](#)
- [Filters in Azure Cognitive Search](#)
- [Search Documents \(Azure Cognitive Search REST API\)](#)
- [Lucene query syntax](#)
- [Simple query syntax in Azure Cognitive Search](#)

OData \$filter syntax in Azure Cognitive Search

10/4/2020 • 7 minutes to read • [Edit Online](#)

Azure Cognitive Search uses [OData filter expressions](#) to apply additional criteria to a search query besides full-text search terms. This article describes the syntax of filters in detail. For more general information about what filters are and how to use them to realize specific query 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)
Logical operators	<code>not</code>
Comparison operators	<code>eq</code> , <code>ne</code> , <code>gt</code> , <code>lt</code> , <code>ge</code> , <code>le</code>
Logical operators	<code>and</code>
Logical operators	<code>or</code>

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:

```
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:

```
not Rating gt 5
```

You'll get this error message:

```
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:

```
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:

```
$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:

```
$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:

```
$filter=LastRenovationDate ge 2010-01-01T00:00:00-08:00
```

Find all hotels that have parking included and where all rooms are non-smoking:

```
$filter=ParkingIncluded and Rooms/all(room: not room/SmokingAllowed)
```

- OR -

```
$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:

```
$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):

```
$filter=Rooms/any(room: room/Tags/any(tag: tag eq 'wifi'))
```

Find all hotels with any rooms:

```
$filter=Rooms/any()
```

Find all hotels that don't have rooms:

```
$filter=not Rooms/any()
```

Find all hotels within 10 kilometers of a given reference point (where `Location` is a field of type `Edm.GeographyPoint`):

```
$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](#).

```
$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:

```
$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:

```
$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 '|'):

```
$filter=search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels where all rooms have the tag 'wifi' or 'tub':

```
$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.

```
$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`.

```
$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`.

```
$filter=search.ismatchscoring('hostel') and rating ge 4 or search.ismatchscoring('motel') and rating eq 5
```

Find documents without the word "luxury".

```
$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.

```
$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](#).

```
$filter=search.ismatch('"hotel airport"~5', 'Description', 'full', 'any') and not Rooms/any(room:room/SmokingAllowed)
```

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 \$orderby syntax in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

You can use the [OData \\$orderby parameter](#) to apply a custom sort order for search results in Azure Cognitive Search. This article describes the syntax of **\$orderby** in detail. For more general information about how to use **\$orderby** when presenting search results, see [How to work with search results in Azure Cognitive Search](#).

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.

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:

```
$orderby=BaseRate asc
```

Sort hotels descending by rating, then ascending by base rate (remember that ascending is the default):

```
$orderby=Rating desc,BaseRate
```

Sort hotels descending by rating, then ascending by distance from the given coordinates:

```
$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:

```
$orderby=search.score() desc,Rating desc,geo.distance(Location, geography'POINT(-122.131577 47.678581)') asc
```

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 \(Azure Cognitive Search REST API\)](#)

OData \$select syntax in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

You can use the [OData \\$select parameter](#) to choose which fields to include in search results from Azure Cognitive Search. This article describes the syntax of `$select` in detail. For more general information about how to use `$select` when presenting search results, 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 sub-fields explicitly, all retrievable sub-fields will be included in the query result set. For example, assume your index has an `Address` field with `Street`, `City`, and `Country` sub-fields that are all retrievable. If you specify `Address` in `$select`, the query results will include all three sub-fields.

Examples

Include the `HotelId`, `HotelName`, and `Rating` top-level fields in the results, as well as the `City` sub-field of `Address`:

```
$select=HotelId, HotelName, Rating, Address/City
```

An example result might look like this:

```
{  
  "HotelId": "1",  
  "HotelName": "Secret Point Motel",  
  "Rating": 4,  
  "Address": {  
    "City": "New York"  
  }  
}
```

Include the `HotelName` top-level field in the results, as well as all sub-fields of `Address`, and the `Type` and `BaseRate` sub-fields of each object in the `Rooms` collection:

```
$select=HotelName, Address, Rooms/Type, Rooms/BaseRate
```

An example result might look like this:

```
{  
  "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 \(Azure Cognitive Search REST API\)](#)

OData collection operators in Azure Cognitive Search - `any` and `all`

10/4/2020 • 3 minutes to read • [Edit Online](#)

When writing an [OData filter expression](#) to use with Azure Cognitive Search, it is 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":

```
tags/any(t: t eq 'wifi')
```

Match documents where every element of the `ratings` field falls between 3 and 5, inclusive:

```
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:

```
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:

```
not rooms/any()
```

Match documents where for all rooms, the `rooms/amenities` field contains "tv" and `rooms/baseRate` is less than 100:

```
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
<code>Collection(Edm.ComplexType)</code>	Everything except <code>search.ismatch</code> and <code>search.ismatchscoring</code>	Same
<code>Collection(Edm.String)</code>	Comparisons with <code>eq</code> or <code>search.in</code> Combining sub-expressions with <code>or</code>	Comparisons with <code>ne</code> or <code>not search.in()</code> Combining sub-expressions with <code>and</code>
<code>Collection(Edm.Boolean)</code>	Comparisons with <code>eq</code> or <code>ne</code>	Same
<code>Collection(Edm.GeographyPoint)</code>	Using <code>geo.distance</code> with <code>lt</code> or <code>le</code> <code>geo.intersects</code> Combining sub-expressions with <code>or</code>	Using <code>geo.distance</code> with <code>gt</code> or <code>ge</code> <code>not geo.intersects(...)</code> Combining sub-expressions with <code>and</code>

DATA TYPE	FEATURES ALLOWED IN LAMBDA EXPRESSIONS WITH ANY	FEATURES ALLOWED IN LAMBDA EXPRESSIONS WITH ALL
<code>Collection(Edm.DateTimeOffset)</code> , <code>Collection(Edm.Double)</code> , <code>Collection(Edm.Int32)</code> , <code>Collection(Edm.Int64)</code>	<p>Comparisons using <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>gt</code>, <code>le</code>, or <code>ge</code></p> <p>Combining comparisons with other sub-expressions using <code>or</code></p> <p>Combining comparisons except <code>ne</code> with other sub-expressions using <code>and</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Disjunctive Normal Form (DNF)</p>	<p>Comparisons using <code>eq</code>, <code>ne</code>, <code>lt</code>, <code>gt</code>, <code>le</code>, or <code>ge</code></p> <p>Combining comparisons with other sub-expressions using <code>and</code></p> <p>Combining comparisons except <code>eq</code> with other sub-expressions using <code>or</code></p> <p>Expressions using combinations of <code>and</code> and <code>or</code> in Conjunctive Normal Form (CNF)</p>

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 \(Azure Cognitive Search REST API\)](#)

OData comparison operators in Azure Cognitive Search - `eq`, `ne`, `gt`, `lt`, `ge`, and `le`

10/4/2020 • 5 minutes to read • [Edit Online](#)

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
<code>Edm.Double</code>	<code>Edm.Double</code>	Comparison is subject to special rules for NaN
<code>Edm.Double</code>	<code>Edm.Int64</code>	Constant is converted to <code>Edm.Double</code> , resulting in a loss of precision for values of large magnitude
<code>Edm.Double</code>	<code>Edm.Int32</code>	n/a
<code>Edm.Int64</code>	<code>Edm.Double</code>	Comparisons to <code>NaN</code> , <code>-INF</code> , or <code>INF</code> are not allowed
<code>Edm.Int64</code>	<code>Edm.Int64</code>	n/a
<code>Edm.Int64</code>	<code>Edm.Int32</code>	Constant is converted to <code>Edm.Int64</code> before comparison
<code>Edm.Int32</code>	<code>Edm.Double</code>	Comparisons to <code>NaN</code> , <code>-INF</code> , or <code>INF</code> are not allowed
<code>Edm.Int32</code>	<code>Edm.Int64</code>	n/a
<code>Edm.Int32</code>	<code>Edm.Int32</code>	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 <code>NULL</code>	RESULT WHEN ONLY THE CONSTANT IS <code>NULL</code>	RESULT WHEN BOTH THE FIELD OR VARIABLE AND THE CONSTANT ARE <code>NULL</code>
<code>gt</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>lt</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>ge</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>le</code>	<code>false</code>	HTTP 400: Bad Request error	HTTP 400: Bad Request error
<code>eq</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>ne</code>	<code>true</code>	<code>true</code>	<code>false</code>

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 <code>NAN</code>
<code>gt</code>	<code>false</code>
<code>lt</code>	<code>false</code>
<code>ge</code>	<code>false</code>
<code>le</code>	<code>false</code>
<code>eq</code>	<code>false</code>
<code>ne</code>	<code>true</code>

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:

```
Rating ge 3 and Rating le 5
```

Match documents where the `Location` field is less than 2 kilometers from the given latitude and longitude:

```
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:

```
LastRenovationDate ge 2015-01-01T00:00:00.000Z
```

Match documents where the `Details/Sku` field is not `null`:

```
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:

```
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`

10/4/2020 • 3 minutes to read • [Edit Online](#)

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
<code>search</code>	<code>Edm.String</code>	The search query (in either simple or full Lucene query syntax).
<code>searchFields</code>	<code>Edm.String</code>	Comma-separated list of searchable fields to search in; defaults to all searchable fields in the index. When using fielded search in the <code>search</code> parameter, the field specifiers in the Lucene query override any fields specified in this parameter.
<code>queryType</code>	<code>Edm.String</code>	<code>'simple'</code> or <code>'full'</code> ; defaults to <code>'simple'</code> . Specifies what query language was used in the <code>search</code> parameter.
<code>searchMode</code>	<code>Edm.String</code>	<code>'any'</code> or <code>'all'</code> , defaults to <code>'any'</code> . Indicates whether any or all of the search terms in the <code>search</code> parameter must be matched in order to count the document as a match. When using the Lucene Boolean operators in the <code>search</code> 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 .
```

```
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.

```
search.ismatchscoring('hostel') and Rating ge 4 or search.ismatchscoring('motel') and Rating eq 5
```

Find documents without the word "luxury".

```
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.

```
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](#).

```
search.ismatch('"hotel airport"~5', 'Description', 'full', 'any') and Rooms/any(room: not room/SmokingAllowed)
```

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 geo-spatial functions in Azure Cognitive Search - `geo.distance` and `geo.intersects`

10/4/2020 • 4 minutes to read • [Edit Online](#)

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`):

```
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](#).

```
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:

```
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:

```
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

10/4/2020 • 3 minutes to read • [Edit Online](#)

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 <code>B</code> IS <code>NULL</code>
<code>b</code>	<code>false</code>
<code>not b</code>	<code>true</code>
<code>b eq true</code>	<code>false</code>
<code>b eq false</code>	<code>false</code>
<code>b eq null</code>	<code>true</code>
<code>b ne true</code>	<code>true</code>
<code>b ne false</code>	<code>true</code>
<code>b ne null</code>	<code>false</code>
<code>b and true</code>	<code>false</code>
<code>b and false</code>	<code>false</code>
<code>b or true</code>	<code>true</code>
<code>b or false</code>	<code>false</code>

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:

```
rating ge 3 and rating le 5
```

Match documents where all elements of the `ratings` field are less than 3 or greater than 5:

```
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".

```
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:

```
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

10/4/2020 • 3 minutes to read • [Edit Online](#)

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:

```
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:

```
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
<code>variable</code>	<code>Edm.String</code>	A string field reference (or a range variable over a string collection field in the case where <code>search.in</code> is used inside an <code>any</code> or <code>all</code> expression).
<code>valueList</code>	<code>Edm.String</code>	A string containing a delimited list of values to match against the <code>variable</code> parameter. If the <code>delimiters</code> parameter is not specified, the default delimiters are space and comma.
<code>delimiters</code>	<code>Edm.String</code>	A string where each character is treated as a separator when parsing the <code>valueList</code> parameter. The default value of this parameter is <code>' , '</code> 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 <code>' '</code> 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:

```
search.in(HotelName, 'Sea View motel,Budget hotel', ',')
```

Find all hotels with name equal to either 'Sea View motel' or 'Budget hotel' separated by '|':

```
search.in(HotelName, 'Sea View motel|Budget hotel', '|')
```

Find all hotels with rooms that have the tag 'wifi' or 'tub':

```
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.

```
Rooms/any(room: room/Tags/any(tag: search.in(tag, 'heated towel racks,hairdryer included', ',')))
```

Find all hotels without the tag 'motel' or 'cabin':

```
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

10/4/2020 • 2 minutes to read • [Edit Online](#)

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 will be used to break ties by default. However, sometimes it is useful to use the relevance score as an initial sort criteria, and some other criteria as the tie-breaker. The `search.score` function allows you to do this.

Syntax

The syntax for `search.score` in `$orderby` is `search.score()`. The function `search.score` does not take any parameters. It can be used with the `asc` or `desc` sort-order specifier, just like any other clause in the `$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:

```
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 \(Azure Cognitive Search EST API\)](#)

OData expression syntax reference for Azure Cognitive Search

10/4/2020 • 3 minutes to read • [Edit Online](#)

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 sub-fields of complex fields everywhere in the API, such as when listing sub-fields 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 sub-fields 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

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'

/* 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](#)

Built-in cognitive skills for text and image processing during indexing (Azure Cognitive Search)

10/4/2020 • 3 minutes to read • [Edit Online](#)

In this article, you learn about the cognitive skills provided with Azure Cognitive Search that you can include in a skillset to extract content and structure. A *cognitive skill* is a module or operation that transforms content in some way. Often, it is a component that extracts data or infers structure, and therefore augments our understanding of the input data. Almost always, the output is text-based. A *skillset* is collection of skills that define the enrichment pipeline.

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](#).

The [incremental enrichment \(preview\)](#) feature allows you to provide a cache that enables the indexer to be more efficient at running only the cognitive skills that are necessary if you modify your skillset in the future, saving you time and money.

Built-in skills

Several skills are flexible in what they consume or produce. In general, most skills are based on pre-trained models, which means you cannot train the model using your own training data. The following table enumerates and describes the skills provided by Microsoft.

SKILL	DESCRIPTION
Microsoft.Skills.Text.CustomEntityLookupSkill	Looks for text from a custom, user-defined list of words and phrases.
Microsoft.Skills.Text.KeyPhraseSkill	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.
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.
Microsoft.Skills.Text.MergeSkill	Consolidates text from a collection of fields into a single field.

SKILL	DESCRIPTION
Microsoft.Skills.Text.EntityRecognitionSkill	This skill uses a pretrained model to establish entities for a fixed set of categories: people, location, organization, emails, URLs, datetime fields.
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.
Microsoft.Skills.Text.SentimentSkill	This skill uses a pretrained model to score positive or negative sentiment on a record by record basis. The score is between 0 and 1. Neutral scores occur for both the null case when sentiment cannot be detected, and for text that is considered neutral.
Microsoft.Skills.Text.SplitSkill	Splits text into pages so that you can enrich or augment content incrementally.
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.
Microsoft.Skills.Vision.ImageAnalysisSkill	This skill uses an image detection algorithm to identify the content of an image and generate a text description.
Microsoft.Skills.Vision.OcrSkill	Optical character recognition.
Microsoft.Skills.Util.ConditionalSkill	Allows filtering, assigning a default value, and merging data based on a condition.
Microsoft.Skills.Util.DocumentExtractionSkill	Extracts content from a file within the enrichment pipeline.
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.)
Microsoft.Skills.Custom.WebApISkill	Allows extensibility of an AI enrichment pipeline by making an HTTP call into a custom Web API
Microsoft.Skills.Custom.AmlSkill	Allows extensibility of an AI enrichment pipeline with an Azure Machine Learning model

For guidance on creating a [custom skill](#), see [How to 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

11/4/2019 • 4 minutes to read • [Edit Online](#)

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 an Azure Cognitive Services API, and you aren't charged for using it. However, you should still [attach a Cognitive Services resource](#) to override the "Free" resource option that limits you to a small number of enrichments per day.

@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
condition	<p>This input is an evaluated field that represents the condition to evaluate. This condition should evaluate to a Boolean value (<code>true</code> or <code>false</code>).</p> <p>Examples:</p> <pre>= true" = \$(/document/language) == 'fr'" = \$(/document/pages/*,/language) == \$(/document/expectedLanguage)"</pre>
whenTrue	<p>This input is an evaluated field that represents the value to return if the condition is evaluated to <code>true</code>. Constants strings should be returned in single quotation marks (' and ').</p> <p>Sample values:</p> <pre>= 'contract'" = \$(/document/contractType)" = \$(/document/entities/*)"</pre>
whenFalse	<p>This input is an evaluated field that represents the value to return if the condition is evaluated to <code>false</code>.</p> <p>Sample values:</p> <pre>= 'contract'" = \$(/document/contractType)" = \$(/document/entities/*)"</pre>

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*.

```
{  
    "@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.

```
{  
    "@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/sentiment/*/sentiment").

```
{  
    "@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*.

```
{  
    "@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](#)

Custom Entity Lookup cognitive skill (Preview)

10/4/2020 • 9 minutes to read • [Edit Online](#)

IMPORTANT

This skill is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). There is currently no portal or .NET SDK support.

The **Custom Entity Lookup** skill looks for text from a custom, user-defined list of words and phrases. Using this list, it labels all documents with any matching entities. The skill also supports a degree of fuzzy matching that can be applied to find matches that are similar but not quite exact.

This skill is not bound to a Cognitive Services API and can be used free of charge during the preview period. You should still [attach a Cognitive Services resource](#), however, to override the daily enrichment limit. The daily limit applies to free access to Cognitive Services when accessed through Azure 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](#).
- The maximum entities definition table supported is 10 MB if it is provided using the *entitiesDefinitionUri* parameter.
- If the entities are defined inline, using the *inlineEntitiesDefinition* parameter, the maximum supported size is 10 KB.

Skill parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
<code>entitiesDefinitionUri</code>	Path to a 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 config must be accessible over HTTPS. See Custom Entity Definition Format " below for expected CSV or JSON schema.
<code>inlineEntitiesDefinition</code>	Inline JSON entity definitions. This parameter supersedes the <code>entitiesDefinitionUri</code> parameter if present. No more than 10 KB of configuration may be provided inline. See Custom Entity Definition below for expected JSON schema.

PARAMETER NAME	DESCRIPTION
<code>defaultLanguageCode</code>	(Optional) Language code of the input text used to tokenize and delineate input text. The following languages are supported: <code>da, de, en, es, fi, fr, it, ko, pt</code> . The default is English (<code>en</code>). If you pass a languagecode-countrycode format, only the languagecode part of the format is used.

Skill inputs

INPUT NAME	DESCRIPTION
<code>text</code>	The text to analyze.
<code>languageCode</code>	Optional. Default is <code>"en"</code> .

Skill outputs

OUTPUT NAME	DESCRIPTION
<code>entities</code>	An array of objects that contain information about the matches that were found, and related metadata. Each of the entities identified may contain the following fields: <ul style="list-style-type: none"> • <code>name</code>: The top-level entity identified. The entity represents the "normalized" form. • <code>id</code>: A unique identifier for the entity as defined by the user in the "Custom Entity Definition Format". • <code>description</code>: Entity description as defined by the user in the "Custom Entity Definition Format". • <code>type</code>: Entity type as defined by the user in the "Custom Entity Definition Format". • <code>subtype</code>: Entity subtype as defined by the user in the "Custom Entity Definition Format". • <code>matches</code>: Collection that describes each of the matches for that entity on the source text. Each match will have the following members: <ul style="list-style-type: none"> • <code>text</code>: The raw text match from the source document. • <code>offset</code>: The location where the match was found in the text. • <code>length</code>: The length of the matched text. • <code>matchDistance</code>: The number of characters different this match was from original entity name or alias.

Custom Entity Definition Format

There are 3 different ways to provide the list of custom entities to the Custom Entity Lookup skill. You can provide the list in a .CSV file, a .JSON file or as an inline definition as part of the skill definition.

If the definition file is a .CSV or .JSON file, the path of the file needs to be provided as part of the `entitiesDefinitionUri` parameter. In this case, the file is downloaded once at the beginning of each indexer run. The file must be accessible as long as the indexer is intended to run. Also, the file must be encoded UTF-8.

If the definition is provided inline, it should be provided as inline as the content of the *inlineEntitiesDefinition* skill parameter.

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 as entities found (Bill Gates, Satya Nadella, Microsoft), but they will be identified if any of the terms on the line (aliases) are matched on the text. For instance, 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:

```
[  
  {  
    "name" : "Bill Gates"  
  },  
  {  
    "name" : "Microsoft"  
  },  
  {  
    "name" : "Satya Nadella"  
  }]
```

A more complex example of a JSON definition can optionally provide the id, description, type and subtype of each entity -- as well as other *aliases*. If an alias term is matched, the entity will be returned as well:

```
[
  {
    "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 in more details the different configuration parameters you can set when defining the entities to match:

FIELD NAME	DESCRIPTION
<code>name</code>	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.
<code>description</code>	(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.
<code>type</code>	(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.
<code>subtype</code>	(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.
<code>id</code>	(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.

FIELD NAME	DESCRIPTION
<code>caseSensitive</code>	(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
<code>fuzzyEditDistance</code>	(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.
<code>defaultCaseSensitive</code>	(Optional) Changes the default case sensitivity value for this entity. It can be used to change the default value of all aliases caseSensitive values.
<code>defaultFuzzyEditDistance</code>	(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.
<code>aliases</code>	(Optional) An array of complex objects that can be used to specify alternative spellings or synonyms to the root entity name.

ALIAS PROPERTIES	DESCRIPTION
<code>text</code>	The alternative spelling or representation of some target entity name.
<code>caseSensitive</code>	(Optional) Acts the same as root entity "caseSensitive" parameter above, but applies to only this one alias.
<code>fuzzyEditDistance</code>	(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 provide the list of custom entities to match inline directly into the skill definition. In that case you can use a similar JSON format to the one described above, but it is inlined in the skill definition. Only configurations that are less than 10 KB in size (serialized size) can be defined inline.

Sample definition

A sample skill definition using an inline format is shown below:

```
{
  "@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, if you decide to provide a pointer to the entities definition file, a sample skill definition using the `entitiesDefinitionUri` format is shown below:

```
{
  "@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 input

```
{
  "values": [
    {
      "recordId": "1",
      "data":
        {
          "text": "The company, Microsoft, was founded by Bill Gates. Microsoft's gaming console is called Xbox",
          "languageCode": "en"
        }
    }
  ]
}
```

Sample output

```
{
  "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
              }
            ]
          }
        ]
      }
    }
  ]
}
```

Errors and warnings

Warning: 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. In this case, we will stop including duplicate matches. If this is unacceptable to you, please file a [support ticket](#) so we can assist you with your individual use case.

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Entity Recognition skill \(to search for well known entities\)](#)

Document Extraction cognitive skill

10/4/2020 • 4 minutes to read • [Edit Online](#)

IMPORTANT

This skill is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). There is currently no portal or .NET SDK support.

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

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 indexing. 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 [pricing page](#).

@odata.type

Microsoft.Skills.Util.DocumentExtractionSkill

Skill parameters

Parameters are case-sensitive.

INPUTS	ALLOWED VALUES	DESCRIPTION
<code>parsingMode</code>	<code>default</code> <code>text</code> <code>json</code>	Set to <code>default</code> for document extraction from files that are not pure text or json. Set to <code>text</code> to improve performance on plain text files. Set to <code>json</code> to extract structured content from json files. If <code>parsingMode</code> is not defined explicitly, it will be set to <code>default</code> .
<code>dataToExtract</code>	<code>contentAndMetadata</code> <code>allMetadata</code>	Set to <code>contentAndMetadata</code> to extract all metadata and textual content from each file. Set to <code>allMetadata</code> to extract only the content-type specific metadata (for example, metadata unique to just .png files). If <code>dataToExtract</code> is not defined explicitly, it will be set to <code>contentAndMetadata</code> .

INPUTS	ALLOWED VALUES	DESCRIPTION
<code>configuration</code>	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
<code>imageAction</code>	<code>none</code> <code>generateNormalizedImages</code> <code>generateNormalizedImagePerPage</code>	<p>Set to <code>none</code> to ignore embedded images or image files in the data set. This is the default.</p> <p>For image analysis using cognitive skills, set to <code>generateNormalizedImages</code> to have the skill create an array of normalized images as part of document cracking. This action requires that <code>parsingMode</code> is set to <code>default</code> and <code>dataToExtract</code> is set to <code>contentAndMetadata</code>. 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.</p> <p>If you set to <code>generateNormalizedImagePerPage</code>, 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 <code>generateNormalizedImages</code> was set.</p>
<code>normalizedImageMaxWidth</code>	Any integer between 50-10000	The maximum width (in pixels) for normalized images generated. The default is 2000.
<code>normalizedImageMaxHeight</code>	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 follows:

```
{
  "$type": "file",
  "data": "BASE64 encoded string of the file"
}
```

This file reference object can be generated one of 3 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 data 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 (i.e. `/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.

Skill outputs

OUTPUT NAME	DESCRIPTION
content	The textual content of the document.
normalized_images	When the <code>imageAction</code> is set to a value other then <code>none</code> , the new <code>normalized_images</code> field will contain an array of images. See the documentation for image extraction for more details on the output format of each image.

Sample definition

```
{
  "@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": "content"
    },
    {
      "name": "normalized_images",
      "targetName": "normalized_images"
    }
  ]
}
```

Sample input

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "file_data": {
          "$type": "file",
          "data": "aGVsbG8="
        }
      }
    }
  ]
}
```

Sample output

```
{
  "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 Recognition cognitive skill

10/4/2020 • 4 minutes to read • [Edit Online](#)

The **Entity Recognition** skill extracts entities of different types from text. This skill uses the machine learning models provided by [Text Analytics](#) in Cognitive Services.

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](#).

Skill parameters

Parameters are case-sensitive and are all optional.

PARAMETER NAME	DESCRIPTION
<code>categories</code>	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.
<code>defaultLanguageCode</code>	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.
<code>minimumPrecision</code>	A value between 0 and 1. If the confidence score (in the <code>namedEntities</code> output) is lower than this value, the entity is not returned. The default is 0.

PARAMETER NAME	DESCRIPTION
<code>includeTypelessEntities</code>	Set to <code>true</code> if you want to recognize well-known entities that don't fit the current categories. Recognized entities are returned in the <code>entities</code> 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 <code>entities</code> output field. Default is <code>false</code>

Skill inputs

INPUT NAME	DESCRIPTION
<code>languageCode</code>	Optional. Default is <code>"en"</code> .
<code>text</code>	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
<code>persons</code>	An array of strings where each string represents the name of a person.
<code>locations</code>	An array of strings where each string represents a location.
<code>organizations</code>	An array of strings where each string represents an organization.
<code>quantities</code>	An array of strings where each string represents a quantity.
<code>dateTimes</code>	An array of strings where each string represents a DateTime (as it appears in the text) value.
<code>urls</code>	An array of strings where each string represents a URL
<code>emails</code>	An array of strings where each string represents an email

OUTPUT NAME	DESCRIPTION
namedEntities	An array of complex types that contains the following fields: <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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) <ul style="list-style-type: none"> ◦ text (the raw text for the entity) ◦ offset (the location where it was found) ◦ length (the length of the raw entity text)

Sample definition

```
{
  "@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

```
{
  "values": [
    {
      "recordId": "1",
      "data":
        {
          "text": "Contoso corporation was founded by John Smith. They can be reached at contact@contoso.com",
          "languageCode": "en"
        }
    }
  ]
}
```

Sample output

```
{
  "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.](#)

Error cases

If the language code for the document is unsupported, an error is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Image Analysis cognitive skill

10/4/2020 • 6 minutes to read • [Edit Online](#)

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 skill uses the machine learning models provided by [Computer Vision](#) in Cognitive Services.

NOTE

Small volumes (under 20 transactions) can be executed for free in Azure Cognitive Search, but larger workloads require [attaching 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.Vision.ImageAnalysisSkill

Skill parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
<code>defaultLanguageCode</code>	A string indicating the language to return. The service returns recognition results in a specified language. If this parameter is not specified, the default value is "en". Supported languages are: <i>en</i> - English (default) <i>es</i> - Spanish <i>ja</i> - Japanese <i>pt</i> - Portuguese <i>zh</i> - Simplified Chinese

PARAMETER NAME	DESCRIPTION
<code>visualFeatures</code>	<p>An array of strings indicating the visual feature types to return. Valid visual feature types include:</p> <ul style="list-style-type: none"> • <i>adult</i> - detects if the image is pornographic in nature (depicts nudity or a sex act), or is gory (depicts extreme violence or blood). Sexually suggestive content (also known as racy content) is also detected. • <i>brands</i> - detects various brands within an image, including the approximate location. The <i>brands</i> visual feature is only available in English. • <i>categories</i> - categorizes image content according to a taxonomy defined in the Cognitive Services Computer Vision documentation. • <i>description</i> - describes the image content with a complete sentence in supported languages. • <i>faces</i> - detects if faces are present. If present, generates coordinates, gender and age. • <i>objects</i> - detects various objects within an image, including the approximate location. The <i>objects</i> visual feature is only available in English. • <i>tags</i> - tags the image with a detailed list of words related to the image content. <p>Names of visual features are case-sensitive. Note that the <i>color</i> and <i>imageType</i> visual features have been deprecated, but this functionality could still be accessed via a custom skill.</p>
<code>details</code>	<p>An array of strings indicating which domain-specific details to return. Valid visual feature types include:</p> <ul style="list-style-type: none"> • <i>celebrities</i> - identifies celebrities if detected in the image. • <i>landmarks</i> - identifies landmarks if detected in the image.

Skill inputs

INPUT NAME	DESCRIPTION
<code>image</code>	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure Blob indexer when <code>imageAction</code> is set to a value other than <code>none</code> . See the sample for more information.

Sample skill definition

```
{
    "description": "Extract image analysis.",
    "@odata.type": "#Microsoft.Skills.Vision.ImageAnalysisSkill",
    "context": "/document/normalized_images/*",
    "defaultLanguageCode": "en",
    "visualFeatures": [
        "tags",
        "categories",
        "description",
        "faces",
        "brands"
    ],
    "inputs": [
        {
            "name": "image",
            "source": "/document/normalized_images/*"
        }
    ],
    "outputs": [
        {
            "name": "categories"
        },
        {
            "name": "tags"
        },
        {
            "name": "description"
        },
        {
            "name": "faces"
        },
        {
            "name": "brands"
        }
    ]
}
```

Sample index (for only the categories, description, faces and tags fields)

```
{
    "fields": [
        {
            "name": "id",
            "type": "Edm.String",
            "key": true,
            "searchable": true,
            "filterable": false,
            "facetable": false,
            "sortable": true
        },
        {
            "name": "blob_uri",
            "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": "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,
```

```
        "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": "name"
        }
    ]
}
]
```

```
        "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": "tags",
        "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
            }
        ]
    }
]
```

Sample output field mapping (for the above index)

```

"outputFieldMappings": [
  {
    "sourceFieldName": "/document/normalized_images/*/categories/*",
    "targetFieldName": "categories"
  },
  {
    "sourceFieldName": "/document/normalized_images/*/tags/*",
    "targetFieldName": "tags"
  },
  {
    "sourceFieldName": "/document/normalized_images/*/description",
    "targetFieldName": "description"
  },
  {
    "sourceFieldName": "/document/normalized_images/*/faces/*",
    "targetFieldName": "faces"
  },
  {
    "sourceFieldName": "/document/normalized_images/*/brands/*/name",
    "targetFieldName": "brands"
  }
]

```

Variation on output field mappings (nested properties)

You can define output field mappings to lower-level properties, such as just landmarks or celebrities. In this case, make sure your index schema has a field to contain landmarks specifically.

```

"outputFieldMappings": [
  {
    "sourceFieldName": "/document/normalized_images/*/categories/detail/celebrities/*",
    "targetFieldName": "celebrities"
  }
]

```

Sample input

```

{
  "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

```

{
  "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
        }
      }
    }
  ]
}

```

```
"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": [
          {
            "name": "Forbidden City",
            "confidence": 0.9978346
          }
        ]
      }
    },
    "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
      }
    ]
  },
  "requestId": "0dbec5ad-a3d3-4f7e-96b4-dfd57efe967d",
  "metadata": {
    "width": 1500,
    "height": 1000,
    "format": "Jpeg"
  },
  "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 is not supported.
InvalidImageUrl	Image URL is badly formatted or not accessible.
InvalidImageFormat	Input data is not a valid image.
InvalidImageSize	Input image is too large.
NotSupportedVisualFeature	Specified feature type is not 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`.

```
"categories": [
  {
    "name":"building_",
    "score":0.97265625,
    "detail":{
      "landmarks":[
        {
          "name":"Forbidden City",
          "confidence":0.92013400793075562
        }
      ]
    }
  }
]
```

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Create Indexer \(REST\)](#)

Key Phrase Extraction cognitive skill

10/4/2020 • 2 minutes to read • [Edit Online](#)

The **Key Phrase Extraction** skill evaluates unstructured text, and for each record, returns a list of key phrases. This skill uses the machine learning models provided by [Text Analytics](#) in Cognitive Services.

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

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.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](#).

Skill parameters

Parameters are case-sensitive.

INPUTS	DESCRIPTION
<code>defaultLanguageCode</code>	(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 Full list of supported languages .
<code>maxKeyPhraseCount</code>	(Optional) The maximum number of key phrases to produce.

Skill inputs

INPUT	DESCRIPTION
<code>text</code>	The text to be analyzed.

INPUT	DESCRIPTION
<code>languageCode</code>	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 Full list of supported languages

Skill outputs

OUTPUT	DESCRIPTION
<code>keyPhrases</code>	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:

```
{
    "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:

```
{
    "@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

```
[  
  "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](#).

Errors and warnings

If you provide an unsupported language code, an error 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

10/4/2020 • 2 minutes to read • [Edit Online](#)

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 by [Text Analytics](#) in Cognitive Services.

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](#)).

Language detection leverages Bing's natural language processing libraries, which exceeds the number of [supported languages and regions](#) listed for Text Analytics. The exact list of languages is not published, but includes all widely-spoken languages, plus variants, dialects, and some regional and cultural languages. If you have content expressed in a less frequently used language, you can [try the Language Detection API](#) to see if it returns a code. The response for languages that cannot be detected is `unknown`.

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.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 inputs

Parameters are case-sensitive.

INPUTS	DESCRIPTION
<code>text</code>	The text to be analyzed.

Skill outputs

OUTPUT NAME	DESCRIPTION
<code>languageCode</code>	The ISO 6391 language code for the language identified. For example, "en".

OUTPUT NAME	DESCRIPTION
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

```
{
  "@odata.type": "#Microsoft.Skills.Text.LanguageDetectionSkill",
  "inputs": [
    {
      "name": "text",
      "source": "/document/text"
    }
  ],
  "outputs": [
    {
      "name": "languageCode",
      "targetName": "myLanguageCode"
    },
    {
      "name": "languageName",
      "targetName": "myLanguageName"
    },
    {
      "name": "score",
      "targetName": "myLanguageScore"
    }
  ]
}
```

Sample input

```
{
  "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."
      }
    }
  ]
}
```

Sample output

```
{  
  "values": [  
    {  
      "recordId": "1",  
      "data":  
        {  
          "languageCode": "en",  
          "languageName": "English",  
          "score": 1,  
        }  
    },  
    {  
      "recordId": "2",  
      "data":  
        {  
          "languageCode": "es",  
          "languageName": "Spanish",  
          "score": 1,  
        }  
    }  
  ]  
}
```

Error cases

If text is expressed in an unsupported language, an error is generated and no language identifier is returned.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

OCR cognitive skill

10/4/2020 • 4 minutes to read • [Edit Online](#)

The **Optical character recognition (OCR)** skill recognizes printed and handwritten text in image files. This skill uses the machine learning models provided by [Computer Vision API v3.0](#) in Cognitive Services. The OCR skill maps to the following functionality:

- For English, Spanish, German, French, Italian, Portuguese, and Dutch, the new "[Read](#)" API is used.
- For all other languages, the "[OCR](#)" API is used.

The OCR skill extracts text from image files. Supported file formats include:

- .JPEG
- .JPG
- .PNG
- .BMP
- .GIF
- .TIFF

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](#).

Skill parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
<code>detectOrientation</code>	Enables autodetection of image orientation. Valid values: true / false.

PARAMETER NAME	DESCRIPTION
<code>defaultLanguageCode</code>	<p>Language code of the input text. Supported languages include:</p> <ul style="list-style-type: none"> zh-Hans (ChineseSimplified) zh-Hant (ChineseTraditional) cs (Czech) da (Danish) nl (Dutch) en (English) fi (Finnish) fr (French) de (German) el (Greek) hu (Hungarian) it (Italian) ja (Japanese) ko (Korean) nb (Norwegian) pl (Polish) pt (Portuguese) ru (Russian) es (Spanish) sv (Swedish) tr (Turkish) ar (Arabic) ro (Romanian) sr-Cyril (SerbianCyrillic) sr-Latn (SerbianLatin) sk (Slovak) unk (Unknown) <p>If the language code is unspecified or null, the language will be set to English. If the language is explicitly set to "unk", the language will be auto-detected.</p>
<code>lineEnding</code>	The value to use between each detected line. Possible values: "Space", "CarriageReturn", "LineFeed". The default is "Space".

Previously, there was a parameter called "textExtractionAlgorithm" for specifying whether the skill should extract "printed" or "handwritten" text. This parameter is deprecated and no longer necessary as the latest Read API algorithm is capable of extracting both types of text at once. If your skill definition already includes this parameter, you do not need to remove it, but it will no longer be used and both types of text will be extracted going forward regardless of what it is set to.

Skill inputs

INPUT NAME	DESCRIPTION
<code>image</code>	Complex Type. Currently only works with "/document/normalized_images" field, produced by the Azure Blob indexer when <code>imageAction</code> is set to a value other than <code>none</code> . See the sample for more information.

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.

Sample definition

```
{
  "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

```
{
  "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.

A common use case for Text Merger is the ability 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 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

```
{
  "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:

```
{  
    //...rest of your indexer definition goes here ...  
    "parameters": {  
        "configuration": {  
            "dataToExtract": "contentAndMetadata",  
            "imageAction": "generateNormalizedImages"  
        }  
    }  
}
```

See also

- [Built-in skills](#)
- [TextMerger skill](#)
- [How to define a skillset](#)
- [Create Indexer \(REST\)](#)

PII Detection cognitive skill

10/4/2020 • 3 minutes to read • [Edit Online](#)

IMPORTANT

This skill is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). There is currently no portal or .NET SDK support.

The **PII Detection** skill extracts personal information from an input text and gives you the option of masking it. This skill uses the machine learning models provided by [Text Analytics](#) in Cognitive Services.

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.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](#).

Skill parameters

Parameters are case-sensitive and all are optional.

PARAMETER NAME	DESCRIPTION
<code>defaultLanguageCode</code>	Language code of the input text. For now, only <code>en</code> is supported.
<code>minimumPrecision</code>	A value between 0.0 and 1.0. If the confidence score (in the <code>piiEntities</code> output) is lower than the set <code>minimumPrecision</code> value, the entity is not returned or masked. The default is 0.0.

PARAMETER NAME	DESCRIPTION
<code>maskingMode</code>	<p>A parameter that provides various ways to mask the personal information detected in the input text. The following options are supported:</p> <ul style="list-style-type: none"> • <code>none</code> (default): No masking occurs and the <code>maskedText</code> output will not be returned. • <code>redact</code>: Removes the detected entities from the input text and does not replace the deleted values. In this case, the offset in the <code>piiEntities</code> output will be in relation to the original text, and not the masked text. • <code>replace</code>: Replaces the detected entities with the character given in the <code>maskingCharacter</code> 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 as well as the output <code>maskedText</code>.
<code>maskingCharacter</code>	<p>The character used to mask the text if the <code>maskingMode</code> parameter is set to <code>replace</code>. The following options are supported: <code>*</code> (default), <code>#</code>, <code>x</code>. This parameter can only be <code>null</code> if <code>maskingMode</code> is not set to <code>replace</code>.</p>

Skill inputs

INPUT NAME	DESCRIPTION
<code>languageCode</code>	Optional. Default is <code>en</code> .
<code>text</code>	The text to analyze.

Skill outputs

OUTPUT NAME	DESCRIPTION
<code>piiEntities</code>	<p>An array of complex types that contains the following fields:</p> <ul style="list-style-type: none"> • <code>text</code> (The actual PII as extracted) • <code>type</code> • <code>subType</code> • <code>score</code> (Higher value means it's more likely to be a real entity) • <code>offset</code> (into the input text) • <code>length</code> <p>Possible types and subTypes can be found here.</p>
<code>maskedText</code>	If <code>maskingMode</code> is set to a value other than <code>none</code> , this output will be the string result of the masking performed on the input text as described by the selected <code>maskingMode</code> . If <code>maskingMode</code> is set to <code>none</code> , this output will not be present.

Sample definition

```
{  
    "@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

```
{  
    "values": [  
        {  
            "recordId": "1",  
            "data":  
                {  
                    "text": "Microsoft employee with ssn 859-98-0987 is using our awesome API's."  
                }  
        }  
    ]  
}
```

Sample output

```
{
  "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 [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.](#)

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](#)

Sentiment cognitive skill

10/4/2020 • 2 minutes to read • [Edit Online](#)

The **Sentiment** skill 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.

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
<code>defaultLanguageCode</code>	(optional) The language code to apply to documents that don't specify language explicitly. See Full list of supported languages

Skill inputs

INPUT NAME	DESCRIPTION
<code>text</code>	The text to be analyzed.
<code>languageCode</code>	(Optional) A string indicating the language of the records. If this parameter is not specified, the default value is "en". See 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

```
{
    "@odata.type": "#Microsoft.Skills.Text.SentimentSkill",
    "inputs": [
        {
            "name": "text",
            "source": "/document/content"
        },
        {
            "name": "languageCode",
            "source": "/document/languagecode"
        }
    ],
    "outputs": [
        {
            "name": "score",
            "targetName": "mySentiment"
        }
    ]
}
```

Sample input

```
{
    "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

```
{
    "values": [
        {
            "recordId": "1",
            "data": {
                "score": 0.01
            }
        }
    ]
}
```

Notes

If empty, a sentiment score is not returned for those records.

Error cases

If a language is not supported, an error is generated and no sentiment score is returned.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Shaper cognitive skill

10/4/2020 • 4 minutes to read • [Edit Online](#)

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

The **Shaper** skill is not bound to a Cognitive Services API and you are not charged for using it. You should still [attach a Cognitive Services resource](#), however, to override the **Free** resource option that limits you to a small number of daily enrichments per day.

@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.

```
{
  "@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.

```
"name": "my-index",
"fields": [
  { "name": "myId", "type": "Edm.String", "key": true, "filterable": true },
  { "name": "analyzedText", "type": "Edm.ComplexType",
    "fields": [
      {
        "name": "text",
        "type": "Edm.String",
        "filterable": false,
        "sortable": false,
        "facetable": false,
        "searchable": true
      },
      {
        "name": "sentiment",
        "type": "Edm.Double",
        "searchable": true,
        "filterable": true,
        "sortable": true,
        "facetable": true
      }
    ]
  }
],
```

Skill input

An incoming JSON document providing usable input for this **Shaper** skill could be:

```
{
  "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.

```
{  
    "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:

```
{  
    "@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.

```
{
  "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:

```
{
  "@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.

```
{  
    "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

10/4/2020 • 2 minutes to read • [Edit Online](#)

The **Text Merge** skill consolidates text from a collection of fields into a single field.

NOTE

This skill is not bound to a Cognitive Services API and you are not charged for using it. You should still [attach a Cognitive Services resource](#), however, to override the **Free** resource option that limits you to a small number of daily enrichments per day.

@odata.type

Microsoft.Skills.Text.MergeSkill

Skill parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
<code>insertPreTag</code>	String to be included before every insertion. The default value is <code>" "</code> . To omit the space, set the value to <code>""</code> .
<code>insertPostTag</code>	String to be included after every insertion. The default value is <code>" "</code> . To omit the space, set the value to <code>""</code> .

Sample input

A JSON document providing usable input for this skill could be:

```
{
  "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 `""`.

```
{  
  "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](#).

```
{
  "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:

```
{  
    //...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

10/4/2020 • 2 minutes to read • [Edit Online](#)

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 is not bound to a Cognitive Services API and you are not charged for using it. You should still [attach a Cognitive Services resource](#), however, to override the **Free** resource option that limits you to a small number of daily enrichments per day.

@odata.type

Microsoft.Skills.Text.SplitSkill

Skill Parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
<code>textSplitMode</code>	Either "pages" or "sentences"
<code>maximumPageLength</code>	If <code>textSplitMode</code> is set to "pages", this refers to the maximum page length as measured by <code>String.Length</code> . The minimum value is 300. If the <code>textSplitMode</code> is set to "pages", 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".
<code>defaultLanguageCode</code>	(optional) One of the following language codes: <code>da, de, en, es, fi, fr, it, ko, pt</code> . Default is English (en). Few things to consider: <ul style="list-style-type: none">• If you pass a languagecode-countrycode format, only the languagecode part of the format is used.• If the language is not in the previous list, the split skill breaks the text at character boundaries.• 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.

Skill Outputs

PARAMETER NAME	DESCRIPTION
textItems	An array of substrings that were extracted.

Sample definition

```
{
    "@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

```
{
  "values": [
    {
      "recordId": "1",
      "data": {
        "text": "This is a 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

```
{
  "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 and the text is split at character boundaries.

See also

- [Built-in skills](#)
- [How to define a skillset](#)

Text Translation cognitive skill

10/4/2020 • 3 minutes to read • [Edit Online](#)

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 is 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 is not guaranteed to stay in the same region as your Azure Cognitive Search or attached Cognitive Services resource.

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.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](#).

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 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 is not specified, the automatic language detection provided by the Translator Text API will be used to determine the from language. See Full list of supported languages .

INPUTS	DESCRIPTION
suggestedFrom	(Optional) The language code to translate documents from when neither the fromLanguageCode input nor the defaultFromLanguageCode parameter are provided, and the automatic language detection is unsuccessful. If the suggestedFrom language is not specified, English (en) will be used as the suggestedFrom language. See 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 is not specified, the defaultToLanguageCode will be used to translate the text. See Full list of supported languages
fromLanguageCode	A string indicating the current language of the text. If this parameter is not specified, the defaultFromLanguageCode (or automatic language detection if the defaultFromLanguageCode is not provided) will be used to translate the text. See Full list of supported languages

Skill outputs

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 are 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

```
{  
    "@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

```
{  
    "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

```
{  
  "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 from or the to language, an error is generated and text is not 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

10/4/2020 • 5 minutes to read • [Edit Online](#)

IMPORTANT

This skill is currently in public preview. Preview functionality is provided without a service level agreement, and is not recommended for production workloads. For more information, see [Supplemental Terms of Use for Microsoft Azure Previews](#). There is currently no .NET SDK support.

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.

PARAMETER NAME	DESCRIPTION
<code>key</code>	(Required for key authentication) The <code>key</code> for the AML service.
<code>resourceId</code>	(Required for token authentication). The Azure Resource Manager resource ID of the AML service. It should be in the format <code>subscriptions/{guid}/resourceGroups/{resource-group-name}/Microsoft.MachineLearningServices/workspaces/{workspace-name}/services/{service_name}</code> .
<code>region</code>	(Optional for token authentication). The <code>region</code> the AML service is deployed in.
<code>timeout</code>	(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, <code>PT60S</code> 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.
<code>degreeOfParallelism</code>	(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 <code>degreeOfParallelism</code> 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

```
{  
    "@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)

```
{  
    "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.

```
{  
    "detected_language_code": "es"  
}
```

Inline shaping sample definition

```
{  
    "@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

```
{  
    "shapedText": { "content": "Este es un contrato en Inglés" }  
}
```

Inline shaping sample output JSON structure

```
{  
    "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

10/4/2020 • 5 minutes to read • [Edit Online](#)

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 payloads are 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
<code>uri</code>	The URI of the Web API to which the <i>JSON</i> payload will be sent. Only https URI scheme is allowed
<code>httpMethod</code>	The method to use while sending the payload. Allowed methods are <code>PUT</code> or <code>POST</code>
<code>httpHeaders</code>	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: <code>Accept</code> , <code>Accept-Charset</code> , <code>Accept-Encoding</code> , <code>Content-Length</code> , <code>Content-Type</code> , <code>Cookie</code> , <code>Host</code> , <code>TE</code> , <code>Upgrade</code> , <code>Via</code>

PARAMETER NAME	DESCRIPTION
<code>timeout</code>	(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, <code>PT60S</code> 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.
<code>batchSize</code>	(Optional) Indicates how many "data records" (see <i>JSON</i> 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
<code>degreeOfParallelism</code>	(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 <code>degreeOfParallelism</code> can be set to a maximum of 10 and a minimum of 1.

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 Web API will have different fields.

Skill outputs

There are no "predefined" outputs for this skill. Depending on the response your Web API will return, add output fields so that they can be picked up from the *JSON* response.

Sample definition

```
{
    "@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)

```
{
  "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 objects in the `values` array need not be in the same order as the objects in the `values` array sent as a request to the Web API. 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.

```
{
  "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** (with `recordId` not in the original request, or with duplicate values) records in the response `values` array, no enrichment will be performed for **those** records.

For cases when the Web API is unavailable or returns a 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](#)

Deprecated cognitive skills in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

This document describes cognitive skills that are considered deprecated. 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.
- End of support: The last day after which the corresponding skill is considered unsupported. 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.

Microsoft.Skills.Text.NamedEntityRecognitionSkill

Last available api version

2017-11-11-Preview

End of support

February 15, 2019

Recommendations

Use [Microsoft.Skills.Text.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 [Entity Recognition Skill](#), 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

Currently, confidence score as a concept is not supported. The `minimumPrecision` parameter exists on the `EntityRecognitionSkill` for future use and for backwards compatibility.

1. *(Required)* Change the `@odata.type` from `"#Microsoft.Skills.Text.NamedEntityRecognitionSkill"` to `"#Microsoft.Skills.Text.EntityRecognitionSkill"`.
2. *(Optional)* If you are making use of the `entities` output, use the `namedEntities` complex collection output from the `EntityRecognitionSkill` instead. You can use the `targetName` in the skill definition to map it to an annotation called `entities`.
3. *(Optional)* If you do not explicitly specify the `categories`, the `EntityRecognitionSkill` 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

```
{
  "@odata.type": "#Microsoft.Skills.Text.NamedEntityRecognitionSkill",
  "categories": [ "Person" ],
  "defaultLanguageCode": "en",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    }
  ]
}
```

(After) EntityRecognition skill definition

```
{
  "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",
  "categories": [ "Person" ],
  "defaultLanguageCode": "en",
  "inputs": [
    {
      "name": "text",
      "source": "/document/content"
    }
  ],
  "outputs": [
    {
      "name": "persons",
      "targetName": "people"
    }
  ]
}
```

- Slightly complicated migration

(Before) NamedEntityRecognition skill definition

```
{
  "@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 definition

```
{  
    "@odata.type": "#Microsoft.Skills.Text.EntityRecognitionSkill",  
    "categories": [ "Person", "Location", "Organization" ],  
    "defaultLanguageCode": "en",  
    "minimumPrecision": 0.1,  
    "inputs": [  
        {  
            "name": "text",  
            "source": "/document/content"  
        }  
    ],  
    "outputs": [  
        {  
            "name": "persons",  
            "targetName": "people"  
        },  
        {  
            "name": "namedEntities",  
            "targetName": "entities"  
        }  
    ]  
}
```

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Entity Recognition Skill](#)

Named Entity Recognition cognitive skill

10/4/2020 • 2 minutes to read • [Edit Online](#)

The **Named Entity Recognition** skill extracts named entities from text. Available entities include the types

`person`, `location` and `organization`.

IMPORTANT

Named entity recognition skill is now discontinued replaced by [Microsoft.Skills.Text.EntityRecognitionSkill](#). Support stopped on February 15, 2019 and the API was removed from the product on May 2, 2019. 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.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](#).

Skill parameters

Parameters are case-sensitive.

PARAMETER NAME	DESCRIPTION
categories	Array of categories that should be extracted. Possible category types: <code>"Person"</code> , <code>"Location"</code> , <code>"Organization"</code> . If no category is provided, all types are returned.
defaultLanguageCode	Language code of the input text. The following languages are supported: <code>de</code> , <code>en</code> , <code>es</code> , <code>fr</code> , <code>it</code>
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: <ul style="list-style-type: none"> 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

```
{
  "@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

```
{
  "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

```
{
  "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
            }
          ]
        }
      }
    ]
}
```

Error cases

If the language code for the document is unsupported, an error is returned and no entities are extracted.

See also

- [Built-in skills](#)
- [How to define a skillset](#)
- [Entity Recognition Skill](#)

Azure Policy built-in definitions for Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

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 the 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)
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
Diagnostic logs in Search services should be enabled	Audit enabling of diagnostic 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	3.0.0

Next steps

- See the built-ins on the [Azure Policy GitHub repo](#).
- Review the [Azure Policy definition structure](#).
- Review [Understanding policy effects](#).

Documentation resources for AI enrichment in Azure Cognitive Search

10/4/2020 • 2 minutes to read • [Edit Online](#)

AI enrichment is an add-on to indexer-based indexing that finds latent information in non-text sources and undifferentiated text, transforming it into full text searchable content in Azure Cognitive Search.

For built-in processing, the pre-trained AI models in Cognitive Services are called internally to perform the analyses. You can also integrate custom models using Azure Machine Learning, Azure Functions, or other approaches.

The following is a consolidated list of the documentation for AI enrichment.

Concepts

- [AI enrichments](#)
- [Skillsets](#)
- [Debug sessions](#)
- [Knowledge stores](#)
- [Projections](#)
- [Incremental enrichment \(reuse of a cached enriched document\)](#)

Hands on walkthroughs

- [Quickstart: Create a cognitive skillset in the Azure portal](#)
- [Tutorial: Enriched indexing with AI](#)
- [Tutorial: Diagnose, repair, and commit changes to your skillset with Debug Sessions](#)

Knowledge stores

- [Quickstart: Create a knowledge store in the Azure portal](#)
- [Create a knowledge store using REST and Postman](#)
- [View a knowledge store with Storage Explorer](#)
- [Connect a knowledge store with Power BI](#)
- [Projection examples \(how to shape and export enrichments\)](#)

Custom skills (advanced)

- [How to define a custom skills interface](#)
- [Example: Create a custom skill using Azure Functions \(and Bing Entity Search APIs\)](#)
- [Example: Create a custom skill using Python](#)
- [Example: Create a custom skill using Form Recognizer](#)
- [Example: Create a custom skill using Azure Machine Learning](#)

How-to guidance

- [Attach a Cognitive Services resource](#)
- [Define a skillset](#)

- reference annotations in a skillset
- Map fields to an index
- Process and extract information from images
- Configure caching for incremental enrichment
- How to rebuild an Azure Cognitive Search index
- Design tips
- Common errors and warnings

Skills reference

- Built-in skills
 - [Microsoft.Skills.Text.KeyPhraseExtractionSkill](#)
 - [Microsoft.Skills.Text.LanguageDetectionSkill](#)
 - [Microsoft.Skills.Text.EntityRecognitionSkill](#)
 - [Microsoft.Skills.Text.MergeSkill](#)
 - [Microsoft.Skills.Text.PIIDetectionSkill](#)
 - [Microsoft.Skills.Text.SplitSkill](#)
 - [Microsoft.Skills.Text.SentimentSkill](#)
 - [Microsoft.Skills.Text.TranslationSkill](#)
 - [Microsoft.Skills.Vision.ImageAnalysisSkill](#)
 - [Microsoft.Skills.Vision.OcrSkill](#)
 - [Microsoft.Skills.Util.ConditionalSkill](#)
 - [Microsoft.Skills.Util.DocumentExtractionSkill](#)
 - [Microsoft.Skills.Util.ShaperSkill](#)
- Custom skills
 - [Microsoft.Skills.Custom.AmlSkill](#)
 - [Microsoft.Skills.Custom.WebApiSkill](#)
- Deprecated skills
 - [Microsoft.Skills.Text.NamedEntityRecognitionSkill](#)

APIs

- REST API
 - [Create Skillset \(api-version=2020-06-30\)](#)
 - [Create Indexer \(api-version=2020-06-30\)](#)

See also

- [Azure Cognitive Search REST API](#)
- [Indexers in Azure Cognitive Search](#)
- [What is Azure Cognitive Search?](#)

Azure Cognitive Search - frequently asked questions (FAQ)

10/4/2020 • 6 minutes to read • [Edit Online](#)

Find answers to commonly asked questions about concepts, code, and scenarios related to Azure Cognitive Search.

Platform

How is Azure Cognitive Search different from full text search in my DBMS?

Azure Cognitive Search supports multiple data sources, [linguistic analysis for many languages](#), [custom analysis for interesting and unusual data inputs](#), search rank controls through [scoring profiles](#), and user-experience features such as typeahead, hit highlighting, and faceted navigation. It also includes other features, such as synonyms and rich query syntax, but those are generally not differentiating features.

Can I pause Azure Cognitive Search service and stop billing?

You cannot pause the service. Computational and storage resources are allocated for your exclusive use when the service is created. It's not possible to release and reclaim those resources on-demand.

Indexing Operations

Move, backup, and restore indexes or index snapshots?

During the development phase, you may want to move your index between search services. For example, you may use a Basic or Free pricing tier to develop your index, and then want to move it to the Standard or higher tier for production use.

Or, you may want to backup an index snapshot to files that can be used to restore it later.

You can do all these things with the `index-backup-restore` sample code in this [Azure Cognitive Search .NET sample repo](#).

You can also [get an index definition](#) at any time using the Azure Cognitive Search REST API.

There is currently no built-in index extraction, snapshot, or backup-restore feature in the Azure portal. However, we are considering adding the backup and restore functionality in a future release. If you want show your support for this feature, cast a vote on [User Voice](#).

Can I restore my index or service once it is deleted?

No, if you delete an Azure Cognitive Search index or service, it cannot be recovered. When you delete an Azure Cognitive Search service, all indexes in the service are deleted permanently. If you delete an Azure resource group that contains one or more Azure Cognitive Search services, all services are deleted permanently.

Recreating resources such as indexes, indexers, data sources, and skillsets requires that you recreate them from code.

To recreate an index, you must re-index data from external sources. For this reason, it is recommended that you retain a master copy or backup of the original data in another data store, such as Azure SQL Database or Cosmos DB.

As an alternative, you can use the `index-backup-restore` sample code in this [Azure Cognitive Search .NET sample repo](#) to back up an index definition and index snapshot to a series of JSON files. Later, you can use the tool and files to restore the index, if needed.

Can I index from SQL Database replicas (Applies to [Azure SQL Database indexers](#))

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 is no guarantee you get all of the data.

Search Operations

Can I search across multiple indexes?

No, this operation is not supported. Search is always scoped to a single index.

Can I restrict search index access by user identity?

You can implement [security filters](#) with `search.in()` filter. The filter composes well with [identity management services like Azure Active Directory\(AAD\)](#) to trim search results based on defined user group membership.

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

The most common case is not 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 does not match the analyzed form of the word in the search index. For more information on query parsing and analysis, see [query architecture](#).

My wildcard searches are slow.

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

Why is the search rank a constant or equal score of 1.0 for every hit?

By default, search results are scored based on the [statistical properties of matching terms](#), and ordered high to low in the result set. However, some query types (wildcard, prefix, regex) always contribute a constant score to the overall document score. This behavior is by design. Azure Cognitive Search imposes a constant score to 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 is 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. Search results based on a partial input are given a constant score to avoid bias towards potentially unexpected matches.

Skillset Operations

Are there any tips or tricks to reduce cognitive services charges on ingestion?

It is understandable that you don't want to execute built-in skills or custom skills more than is absolutely necessary, especially if you are dealing with millions of documents to process. With that in mind, we have added "incremental enrichment" capabilities to skillset execution. In essence, you can provide a cache location (a blob storage connection string) that will be used to store the output of "intermediate" enrichment steps. That allows the enrichment pipeline to be smart and apply only enrichments that are necessary when you modify your skillset. This will naturally also save indexing time as the pipeline will be more efficient.

Learn more about [incremental enrichment](#)

Design patterns

What is the best approach for implementing localized search?

Most customers choose dedicated fields over a collection when it comes to supporting different locales (languages) in the same index. Locale-specific fields make it possible to assign an appropriate analyzer. For example, assigning the Microsoft French Analyzer to a field containing French strings. It also simplifies filtering. If you know a query is initiated on a fr-fr page, you could limit search results to this field. Or, create a [scoring profile](#) to give the field more relative weight. Azure Cognitive Search supports over [50 language analyzers](#) to choose from.

Next steps

Is your question about a missing feature or functionality? Request the feature on the [User Voice web site](#).

See also

[StackOverflow: Azure Cognitive Search](#)

[How full text search works in Azure Cognitive Search](#)

[What is Azure Cognitive Search?](#)