

4.1 - Create an Azure AI Search solution

- [Overview](#)
- [Introduction](#)
- [Azure AI Search](#)
- [Manage capacity](#)
 - [Service tiers and capacity management](#)
 - [Replicas and partitions](#)
- [Understand search components](#)
 - [Data source](#)
 - [Skillset](#)
 - [Indexer](#)
 - [Index](#)
- [Understand the indexing process](#)
- [Search an index](#)
 - [Full text search](#)
- [Apply filtering and sorting](#)
 - [Filtering results](#)
 - [Filtering with facets](#)
 - [Sorting results](#)
- [Enhance the index](#)
 - [Search-as-you-type](#)
 - [Custom scoring and result boosting](#)
 - [Synonyms](#)
- [Exercise - Create a search solution](#)
 - [Create Azure resources](#)
 - [Create an Azure AI Search resource](#)
 - [Create an Azure AI Services resource](#)
 - [Create a storage account](#)
 - [Prepare to develop an app in Visual Studio Code](#)
 - [Upload Documents to Azure Storage](#)
 - [Index the documents](#)
 - [Search the index](#)
 - [Explore and modify definitions of search components](#)
 - [Get the endpoint and key for your Azure AI Search resource](#)
 - [Review and modify the skillset](#)
 - [Review and modify the index](#)
 - [Review and modify the indexer](#)
 - [Use the REST API to update the search solution](#)
 - [Query the modified index](#)
 - [Create a search client application](#)
 - [Get the endpoint and keys for your search resource](#)
 - [Prepare to use the Azure AI Search SDK](#)

- [Explore code to search an index](#)
 - [Explore code to render search results](#)
 - [Run the web app](#)
 - [More information](#)
 - [Knowledge Check](#)
 - [Summary](#)
-

Overview

Unlock the hidden insights in your data with Azure AI Search.

In this module you'll learn how to:

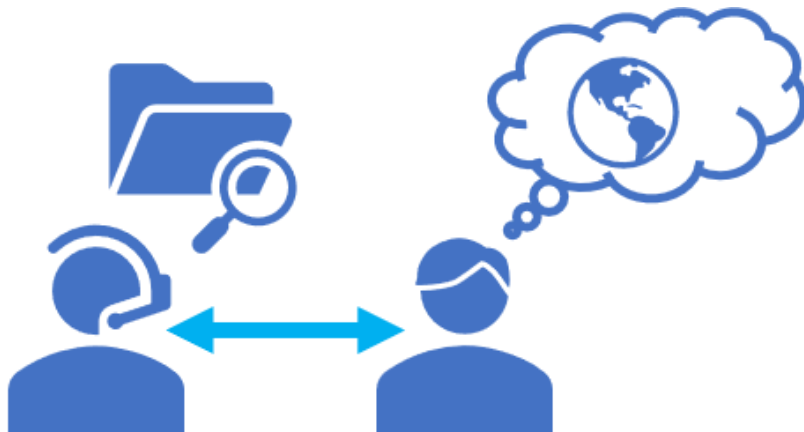
- Create an Azure AI Search solution
 - Develop a search application
-

Introduction

All organizations rely on information to make decisions, answer questions, and function efficiently. The problem for most organizations is not a lack of information, but the **challenge of finding and extracting the information from the massive set of documents**, databases, and other sources in which the information is stored.

For example, suppose *Margie's Travel* is a travel agency that specializes in organizing trips to cities around the world. Over time, the company has amassed a huge amount of information in documents such as brochures, as well as reviews of hotels submitted by customers.

This data is a valuable source of insights for travel agents and customers as they plan trips, but the sheer volume of data can make it difficult to find relevant information to answer a specific customer question.



To address this challenge, Margie's Travel can implement a solution in which the documents are indexed and made easy to search. This solution enables agents and customers to query the index to find relevant documents and extract information from them.

Azure AI Search

Azure AI Search provides a cloud-based solution for indexing and querying a wide range of data sources, and creating comprehensive and high-scale search solutions. With Azure AI Search, you can:

- Index documents and data from a range of sources.
- Use cognitive skills to enrich index data.
- Store extracted insights in a knowledge store for analysis and integration.

By the end of this module, you'll learn how to:

- Create an Azure AI Search solution
- Develop a search application

Manage capacity

To create an Azure AI Search solution, you need to create an **Azure AI Search** resource in your Azure subscription. Depending on the specific solution you intend to build, you may also need Azure resources for data storage and other application services.

Service tiers and capacity management

When you create an Azure AI Search resource, you must specify a *pricing tier*. The pricing tier you select determines the capacity limitations of your search service and the configuration options available to you, as well as the cost of the service. The available pricing tiers are:

- **Free (F)**. Use this tier to explore the service or try the tutorials in the product documentation.
- **Basic (B)**: Use this tier for small-scale search solutions that include a **maximum of 15 indexes and 2 GB of index data**.
- **Standard (S)**: Use this tier for enterprise-scale solutions. There are multiple variants of this tier, including **S**, **S2**, and **S3**; which offer increasing capacity in terms of indexes and storage, and **S3HD**, which is optimized for fast read performance on smaller numbers of indexes.
- **Storage Optimized (L)**: Use a storage optimized tier (**L1** or **L2**) when you need to create large indexes, at the cost of higher query latency.

Note: It's important to select the most suitable pricing tier for your solution, because you **can't change it later**. If you find that the pricing tier you have chosen is no longer suitable for your solution, you **must create a new Azure AI Search resource and recreate all indexes and objects**.

Replicas and partitions

Depending on the pricing tier you select, you can optimize your solution for **scalability and availability** by creating *replicas* and *partitions*.

- *Replicas* are instances of the search service - you can think of them as nodes in a cluster. Increasing the number of replicas can help ensure there is **sufficient capacity to service multiple concurrent query requests while managing ongoing indexing operations**.
- *Partitions* are used to divide an index into multiple storage locations, enabling you to split I/O operations such as querying or rebuilding an index.

The combination of replicas and partitions you configure determines the *search units* used by your solution. Put simply, the number of search units is the number of replicas multiplied by the number of

partitions ($R \times P = SU$). For example, a resource with four replicas and three partitions is using 12 search units.

Tip: You can learn more about pricing tiers and capacity management in the [Azure AI Search documentation](#).

Understand search components

An AI Search solution consists of multiple components, each playing an important part in the process of extracting, enriching, indexing, and searching data.

Data source



Most search solutions start with a *data source* containing the data you want to search. Azure AI Search supports multiple types of data source, including:

- Unstructured files in Azure blob storage containers.
- Tables in Azure SQL Database.
- Documents in Cosmos DB.

Azure AI Search can pull data from these data sources for indexing.

Alternatively, applications **can push JSON data directly into an index**, without pulling it from an existing data store.

Skillset



In a basic search solution, you might index the data extracted from the data source. The information that can be extracted depends on the data source. For example, when indexing data in a database, the fields in the database tables might be extracted; or when indexing a set of documents, file metadata such as file name, modified date, size, and author might be extracted along with the text content of the document.

While a basic search solution that indexes data values extracted directly from the data source can be useful, the expectations of modern application users have driven a need for richer insights into the data.

In Azure AI Search, you can apply **artificial intelligence (AI) skills as part of the indexing process to enrich the source data with new information, which can be mapped to index fields**. The skills used

by an indexer are encapsulated in a *skillset* that defines an **enrichment pipeline** in which each step enhances the source data with insights obtained by a specific AI skill.

Examples of the kind of information that can be extracted by an AI skill include:

- The language in which a document is written.
- Key phrases that might help determine the main themes or topics discussed in a document.
- A sentiment score that quantifies how positive or negative a document is.
- Specific locations, people, organizations, or landmarks mentioned in the content.
- AI-generated descriptions of images, or image text extracted by optical character recognition.
- Custom skills that you develop to meet specific requirements.

Indexer



The *indexer* is the engine that drives the overall indexing process. It takes the outputs extracted using the skills in the skillset, along with the data and metadata values extracted from the original data source, and **maps them to fields in the index**.

An indexer is automatically run when it is created, and can be **scheduled to run at regular intervals or run on demand to add more documents to the index**. In some cases, such as **when you add new fields to an index or new skills to a skillset, you may need to reset the index before re-running the indexer**.

Index



The index is the **searchable result** of the indexing process. It consists of a **collection of JSON documents**, with fields that contain the values extracted during indexing. Client applications can query the index to retrieve, filter, and sort information.

Each index field can be configured with the following attributes:

- **key**: Fields that define a unique key for index records.
- **searchable**: Fields that can be queried using full-text search.
- **filterable**: Fields that can be included in filter expressions to return only documents that match specified constraints.

- **sortable**: Fields that can be used to order the results.
 - **facettable**: Fields that can be used to determine values for *facets* (user interface elements used to filter the results based on a list of known field values).
 - **retrievable**: Fields that can be included in search results (*by default, all fields are retrievable unless this attribute is explicitly removed*).
-

Understand the indexing process

The indexing process works by creating a **document** for each indexed entity. During indexing, an *enrichment pipeline* iteratively builds the documents that combine metadata from the data source with enriched fields extracted by cognitive skills. **You can think of each indexed document as a JSON structure**, which initially consists of a **document** with the index fields you have mapped to fields extracted directly from the source data, like this:

- **document**
 - **metadata_storage_name**
 - **metadata_author**
 - **content**

When the documents in the data source contain images, you can configure the indexer to extract the image data and place each image in a **normalized_images** collection, like this:

- **document**
 - **metadata_storage_name**
 - **metadata_author**
 - **content**
 - **normalized_images**
 - **image0**
 - **image1**

Normalizing the image data in this way enables you to use the collection of images as an input for skills that extract information from image data.

Each skill adds fields to the **document**, so for example a skill that detects the *language* in which a document is written might store its output in a **language** field, like this:

- **document**
 - **metadata_storage_name**
 - **metadata_author**
 - **content**
 - **normalized_images**
 - **image0**
 - **image1**
 - **language**

The document is structured hierarchically, and the skills are applied to a specific *context* within the hierarchy, enabling you to run the skill for each item at a particular level of the document. For example, you

could run an optical character recognition (*OCR*) skill for each image in the normalized images collection to extract any text they contain:

- **document**
 - **metadata_storage_name**
 - **metadata_author**
 - **content**
 - **normalized_images**
 - **image0**
 - **Text**
 - **image1**
 - **Text**
 - **language**

The output fields from each skill can be used as inputs for other skills later in the pipeline, which in turn store *their* outputs in the document structure. For example, we could use a ***merge* skill to combine the original text content with the text extracted from each image** to create a new **merged_content** field that contains all of the text in the document, including image text.

- **document**
 - **metadata_storage_name**
 - **metadata_author**
 - **content**
 - **normalized_images**
 - **image0**
 - **Text**
 - **image1**
 - **Text**
 - **language**
 - **merged_content**

The fields in the final document structure at the end of the pipeline are **mapped to index fields by the indexer** in one of two ways:

1. Fields extracted directly from the source data are all mapped to index fields. These mappings can be *implicit* (fields are automatically mapped to in fields with the same name in the index) or *explicit* (a mapping is defined to match a source field to an index field, often to rename the field to something more useful or to apply a function to the data value as it is mapped).
2. Output fields from the skills in the skillset are explicitly mapped from their hierarchical location in the output to the target field in the index.

Search an index

After you have created and populated an index, you can query it to search for information in the indexed document content. **While you could retrieve index entries based on simple field value matching, most search solutions use *full text search* semantics to query an index.**

Full text search

Full text search describes search solutions that parse text-based document contents to find query terms. Full text search queries in Azure AI Search are based on the *Lucene* query syntax, which provides a rich set of query operations for searching, filtering, and sorting data in indexes.

Azure AI Search supports two variants of the Lucene syntax:

- **Simple** - An intuitive syntax that makes it easy to perform basic searches that match literal query terms submitted by a user.
 - More info here: [Simple query syntax - Azure AI Search | Microsoft Learn](#)
- **Full** - An extended syntax that supports **complex filtering**, **regular expressions**, and other more sophisticated queries.
 - More info here: [Lucene query syntax - Azure AI Search | Microsoft Learn](#)

Client applications submit queries to Azure AI Search by specifying a search expression along with other parameters that determine how the expression is evaluated and the results returned. Some common parameters submitted with a query include:

- **search** - A search expression that includes the terms to be found.
- **queryType** - The Lucene syntax to be evaluated (*simple* or *full*).
- **searchFields** - The index fields to be searched.
- **select** - The fields to be included in the results.
- **searchMode** - Criteria for including results based on multiple search terms. For example, suppose you search for *comfortable hotel*. A searchMode value of *Any* returns documents that contain "comfortable", "hotel", or both; while a searchMode value of *All* restricts results to documents that contain both "comfortable" and "hotel".
- Example:

```
POST /indexes/hotels-sample-index/docs/search?api-version=2023-11-01
{
  "queryType": "full",
  "search": "category:budget AND \"recently renovated\"^3",
  "searchMode": "all"
}
```

Query processing consists of four stages:

1. *Query parsing*. The search expression is evaluated and **reconstructed as a tree of appropriate subqueries**. Subqueries might include *term queries* (finding specific individual words in the search expression - for example *hotel*), *phrase queries* (finding multi-term phrases specified in quotation marks in the search expression - for example, *"free parking"*), and *prefix queries* (finding terms with a specified prefix - for example *air**, which would match *airway*, *air-conditioning*, and *airport*).
2. *Lexical analysis* - The query terms are analyzed and refined based on linguistic rules. For example, text is converted to lower case and nonessential *stopwords* (such as "the", "a", "is", and so on) are removed. Then words are converted to their *root* form (for example, "comfortable" might be simplified to "comfort") and composite words are split into their constituent terms.
3. *Document retrieval* - **The query terms are matched against the indexed terms, and the set of matching documents is identified.**

4. **Scoring** - A relevance score is assigned to each result based on a term frequency/inverse document frequency (**TF/IDF**) **calculation**.

Supplement info from ChatGPT on **Document Scoring**:

- **Aggregate Scores:** For each document, the TF-IDF scores of the query terms are added up to get an overall relevance score for the document with respect to the query.
- **Ranking:** Documents are then ranked based on their relevance scores. The higher the aggregate TF-IDF score, the more relevant the document is considered to be in relation to the query.
- For example, if a user types a query "learn python programming," the search engine calculates the TF-IDF scores of "learn," "python," and "programming" across all documents. Documents that score the highest are those where these terms are more prominent and rare across other documents, making them potentially more relevant to the user's query.

Note: For more information about querying an index, and details about **simple** and **full** syntax, see [Query types and composition in Azure AI Search](#) in the Azure AI Search documentation.

Apply filtering and sorting

It's common in a search solution for users to want to refine query results by filtering and sorting based on field values. Azure AI Search supports both of these capabilities through the search query API.

Filtering results

You can apply filters to queries in two ways:

- By including filter criteria in a *simple search* expression.
- By providing an OData filter expression as a **\$filter** parameter with a *full* syntax search expression.

OData, or Open Data Protocol, is a web-based protocol designed to facilitate the querying and manipulation of data using RESTful APIs. It is a standardized method for building and consuming RESTful APIs in a simple and consistent way. OData filters are a part of this protocol that allow clients to filter the data they retrieve from an API.

You can apply a filter to any *filterable* field in the index.

For example, suppose you want to find documents containing the text *London* that have an **author** field value of *Reviewer*.

You can achieve this result by submitting the following *simple search* expression:

```
search=London+author='Reviewer'  
queryType=Simple
```

Alternatively, you can use an OData filter in a **\$filter** parameter with a *full* Lucene search expression like this:

```
search=London  
$filter=author eq 'Reviewer'
```

```
queryType=Full
```

Tip: OData **\$filter** expressions are case-sensitive!

Filtering with facets

Facets are a useful way to present users with filtering criteria based on field values in a result set. They work best when a field has a small number of discrete values that can be displayed as links or options in the user interface.

To use facets, you must specify *facetable* fields for which you want to retrieve the possible values in an initial query. For example, you could use the following parameters to return all of the possible values for the **author** field:

```
search=*\nfacet=author
```

The results from this query include a collection of discrete facet values that you can display in the user interface for the user to select. Then in a subsequent query, you can use the selected facet value to filter the results:

```
search=*\n$filter=author eq 'selected-facet-value-here'
```

Sorting results

By default, results are sorted based on the relevancy score assigned by the query process, with the highest scoring matches listed first. However, you can override this sort order by including an OData **orderby** parameter that specifies one or more *sortable* fields and a sort order (*asc* or *desc*).

For example, to sort the results so that the most recently modified documents are listed first, you could use the following parameter values:

```
search=*\n$orderby=last_modified desc
```

Note: For more information about using filters, see [Filters in Azure AI Search](#). For information about working with results, including sorting and hit highlighting, see [How to work with search results in Azure AI Search](#).

Enhance the index

With a basic index and a client that can submit queries and display results, you can achieve an effective search solution. However, Azure AI Search supports several ways to enhance an index to provide a better user experience. This topic describes some of the ways in which you can extend your search solution.

Search-as-you-type

By adding a *suggester* to an index, you can enable two forms of search-as-you-type experience to help users find relevant results more easily:

- *Suggestions* - retrieve and display a list of suggested results as the user types into the search box, without needing to submit the search query.
- *Autocomplete* - complete partially typed search terms based on values in index fields.

To implement one or both of these capabilities, create or update an index, defining a **suggester** for one or more fields.

After you've added a suggester, you can use the **suggestion** and **autocomplete** REST API endpoints or the `.NET DocumentsOperationsExtensions.Suggest` and `DocumentsOperationsExtensions.Autocomplete` methods to **submit a partial search term and retrieve a list of suggested results or autocompleted terms to display in the user interface.**

Note: For more information about suggesters, see [Add autocomplete and suggestions to client apps](#) in the Azure AI Search documentation.

Custom scoring and result boosting

By default, search results are sorted by a relevance score that is calculated based on a term-frequency/inverse-document-frequency (TF/IDF) algorithm.

You can **customize** the way this score is calculated by defining a *scoring profile* that applies a **weighting value to specific fields** - essentially increasing the search score for documents when the search term is found in those fields.

Additionally, you can *boost* results based on field values - for example, increasing the relevancy score for documents based on **how recently they were modified or their file size.**

After you've defined a **scoring profile**, you can specify its use in an individual search, or you can modify an index definition so that it uses your custom scoring profile by default.

Note: For more information about scoring profiles, see [Scoring Profiles](#) in the Azure AI Search documentation.

Synonyms

Often, the same thing can be referred to in multiple ways. For example, someone searching for information about the United Kingdom might use any of the following terms:

- United Kingdom
- UK
- Great Britain*
- GB*

**To be accurate, the UK and Great Britain are different entities - but they're commonly confused with one another; so it's reasonable to assume that someone searching for "United Kingdom" might be interested in results that reference "Great Britain".*

To help users find the information they need, you can **define synonym maps that link related terms together.** You can then apply those synonym maps to individual fields in an index, so that when a user

searches for a particular term, documents with fields that contain the term or any of its synonyms will be included in the results.

Note: For more information about synonym maps, see [Synonyms in Azure AI Search](#) in the Azure AI Search documentation.

Exercise - Create a search solution

All organizations rely on information to make decisions, answer questions, and function efficiently. The problem for most organizations is not a lack of information, but the challenge of finding and extracting the information from the massive set of documents, databases, and other sources in which the information is stored.

For example, suppose *Margie's Travel* is a travel agency that specializes in organizing trips to cities around the world. Over time, the company has amassed a huge amount of information in documents such as brochures, as well as reviews of hotels submitted by customers. This data is a valuable source of insights for travel agents and customers as they plan trips, but the sheer volume of data can make it difficult to find relevant information to answer a specific customer question.

To address this challenge, Margie's Travel can use Azure AI Search to implement a solution in which the documents are indexed and enriched by using AI skills to make them easier to search.

Create Azure resources

The solution you will create for Margie's Travel requires the following resources in your Azure subscription:

- An **Azure AI Search** resource, which will manage indexing and querying.
- An **Azure AI Services** resource, which provides AI services for skills that your search solution can use to enrich the data in the data source with AI-generated insights.
- A **Storage account** with a blob container in which the documents to be searched are stored.

Important: Your Azure AI Search and Azure AI Services resources **must be in the same location!**

Create an Azure AI Search resource

1. In a web browser, open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select the **+ Create a resource** button, search for *search*, and create an **Azure AI Search** resource with the following settings:
 - **Subscription:** *Your Azure subscription*
 - **Resource group:** *Create a new resource group (if you are using a restricted subscription, you may not have permission to create a new resource group - use the one provided)*
 - **Service name:** *Enter a unique name*
 - **Location:** *Select a location - note that your Azure AI Search and Azure AI Services resources must be in the same location*

- **Pricing tier:** Basic

[Home](#) > [Azure AI services](#) | [AI Search](#) >

Create a search service ...

[Basics](#)
[Scale](#)
[Networking](#)
[Tags](#)
[Review + create](#)

Project details

Subscription *

Subscription 1

Resource Group *

AI-LEARN-1

[Create new](#)

Instance Details

Service name * ⓘ

klaisearch1

Location *

East US

Pricing tier * ⓘ

Basic
15 GB/Partition, max 3 replicas, max 3 partitions, max 9 search units
[Change Pricing Tier](#)

3. Wait for deployment to complete, and then go to the deployed resource.
4. Review the **Overview** page on the blade for your Azure AI Search resource in the Azure portal. Here, you can use a visual interface to create, test, manage, and monitor the various components of a search solution; including data sources, indexes, indexers, and skillsets.

Create an Azure AI Services resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Services** resource. Your search solution will use this to enrich the data in the datastore with AI-generated insights.

1. Return to the home page of the Azure portal, and then select the **+ Create a resource** button, search for *Azure AI Services*, and create an **Azure AI Services** resource with the following settings:
 - **Subscription:** *Your Azure subscription*
 - **Resource group:** *The same resource group as your Azure AI Search resource*
 - **Region:** *The same location as your Azure AI Search resource*
 - **Name:** *Enter a unique name*
 - **Pricing tier:** Standard S0
2. Select the required checkboxes and create the resource.
3. Wait for deployment to complete, and then view the deployment details.

Create a storage account

1. Return to the home page of the Azure portal, and then select the **+ Create a resource** button, search for *storage account*, and create a **Storage account** resource with the following settings:
 - **Subscription:** *Your Azure subscription*
 - **Resource group:** **The same resource group as your Azure AI Search and Azure AI Services resources*
 - **Storage account name:** *Enter a unique name*
 - **Region:** *Choose any available region*
 - **Performance:** Standard

- **Replication:** Locally-redundant storage (LRS)

[Home](#) > [All resources](#) > [Create a resource](#) > [Marketplace](#) > [Storage account](#) >

Create a storage account ...

Project details

Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.

Subscription *	Subscription 1
Resource group *	AI-LEARN-1

[Create new](#)

Instance details

Storage account name * ⓘ	aistorage1
	✖ The storage account name 'aistorage1' is already taken.
Region * ⓘ	(US) East US
	Deploy to an Azure Extended Zone
Performance * ⓘ	<input checked="" type="radio"/> Standard: Recommended for most scenarios (general-purpose v2 account) <input type="radio"/> Premium: Recommended for scenarios that require low latency.
Redundancy * ⓘ	Locally-redundant storage (LRS)

- On the **Advanced** tab, check the box next to **_Allow enabling anonymous access on individual containers**

Create a storage account ...

✖ Basics Advanced Networking Data protection Encryption Tags Review + create

Security

Configure security settings that impact your storage account.

Require secure transfer for REST API operations ⓘ	<input checked="" type="checkbox"/>
Allow enabling anonymous access on individual containers ⓘ	<input checked="" type="checkbox"/>
Enable storage account key access ⓘ	<input checked="" type="checkbox"/>

2. Wait for deployment to complete, and then go to the deployed resource.
3. On the **Overview** page, note the **Subscription ID** - this identifies the subscription in which the storage account is provisioned.
4. On the **Access keys** page, note that two keys have been generated for your storage account. Then select **Show keys** to view the keys.

Tip: Keep the **Storage Account** blade open - you will need the subscription ID and one of the keys in the next procedure.

Prepare to develop an app in Visual Studio Code

You'll develop your search app using Visual Studio Code. The code files for your app have been provided in a GitHub repo.

Tip: If you have already cloned the **mslearn-knowledge-mining** repo, open it in Visual Studio code. Otherwise, follow these steps to clone it to your development environment.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLearning/mslearn-knowledge-mining` repository to a local folder (it doesn't matter which folder).
3. When the repository has been cloned, open the folder in Visual Studio Code.
4. Wait while additional files are installed to support the C# code projects in the repo.

Note: If you are prompted to add required assets to build and debug, select **Not Now**.

Upload Documents to Azure Storage

Now that you have the required resources, you can upload some documents to your Azure Storage account.

1. In Visual Studio Code, in the **Explorer** pane, expand the **Labfiles\01-azure-search** folder and select **UploadDocs.cmd**.
2. Edit the batch file to replace the **YOUR_SUBSCRIPTION_ID**, **YOUR_AZURE_STORAGE_ACCOUNT_NAME**, and **YOUR_AZURE_STORAGE_KEY** placeholders with the appropriate subscription ID, Azure storage account name, and Azure storage account key values for the storage account you created previously.
3. Save your changes, and then right-click the **01-azure-search** folder and open an integrated terminal.
4. Enter the following command to sign into your Azure subscription by using the Azure CLI.
`az login`
A web browser tab will open and prompt you to sign into Azure. Do so, and then close the browser tab and return to Visual Studio Code.
5. Enter the following command to run the batch file. This will create a blob container in your storage account and upload the documents in the **data** folder to it.
`UploadDocs`

Index the documents

Now that you have the documents in place, you can create a search solution by indexing them.

1. In the Azure portal, browse to your Azure AI Search resource. Then, on its **Overview** page, select **Import data**.
2. On the **Connect to your data** page, in the **Data Source** list, select **Azure Blob Storage**. Then complete the data store details with the following values:
 - **Data Source:** Azure Blob Storage
 - **Data source name:** margies-data
 - **Data to extract:** Content and metadata
 - **Parsing mode:** Default
 - **Connection string:** Select **Choose an existing connection**. Then select your storage account, and finally select the **margies** container that was created by the `UploadDocs.cmd` script.
 - **Managed identity authentication:** None
 - **Container name:** margies

- **Blob folder:** *Leave this blank*
- **Description:** Brochures and reviews in Margie's Travel web site.

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 external data source. Azure AI 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
Data source name *	margies-data
Data to extract ⓘ	Content and metadata
Parsing mode	Default
Subscription	Subscription 1
Connection string *	DefaultEndpointsProtocol=https;AccountName=klaistorage1;A... Choose an existing connection
Managed identity authentication ⓘ	<input checked="" type="radio"/> None <input type="radio"/> System-assigned <input type="radio"/> User-assigned
Container name * ⓘ	margies
Blob folder ⓘ	your/folder/here
Description	Brochures and reviews in Margie's Travel web site.


Next: [Add cognitive skills \(Optional\)](#)

[Give feedback](#)

3. Proceed to the next step (*Add cognitive skills*).

4. In the **Attach Azure AI Services** section, select your Azure AI Services resource.

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

 Enrich and extract structure from your documents through cognitive skills using the same AI algorithms that power AI 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 AI Services

To power your cognitive skills, select an existing AI Services resource or create a new one. The AI Services resource should be in the same region as your search service. If you decide to include cognitive skills in Azure AI Search, please note that their costs are billed separately. By choosing to add these skills, you acknowledge that you are aware of the [Pay-as-you-go pricing](#) for each skill and the [documentation that explains how each skill functions](#). [Learn more](#)

AI Services Resource Name	Region
Free (Limited enrichments)	
AI-LEARN-KL-1	eastus
Refresh	

[Create new AI service](#)

5. In the **Add enrichments** section:

- Change the **Skillset name** to **margies-skillset**.
- Select the option **Enable OCR and merge all text into merged_content field**.
- Ensure that the **Source data field** is set to **merged_content**.
- Leave the **Enrichment granularity level** as **Source field**, which is set the entire contents of the document being indexed; but note that you can change this to extract information at more granular levels, like pages or sentences.
- Select the following enriched fields:
 |Cognitive Skill|Parameter|Field name|
 |---|---|---|
 |Extract location names| |locations|
 |Extract key phrases| |keyphrases|

Detect language		language
Generate tags from images		imageTags
Generate captions from images		imageCaption

^ Add enrichments

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

Skillset name * ⓘ
 margies-skillset ✓

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

Source data field *
 merged_content

Enrichment granularity level ⓘ
 Source field (default)

☐ Enable incremental enrichment ⓘ

Checked items below require a field name.

<input checked="" type="checkbox"/> Text Cognitive Skills	Parameter	Field name
<input type="checkbox"/> Extract people names		people
<input type="checkbox"/> Extract organization names		organizations
<input checked="" type="checkbox"/> Extract location names		locations ✓
<input checked="" type="checkbox"/> Extract key phrases		keyphrases ✓
<input checked="" type="checkbox"/> Detect language		language ✓
<input type="checkbox"/> Translate text	Target Language English	translated_text
<input type="checkbox"/> Extract personally identifiable information		pii_entities

- Double-check your selections (it can be difficult to change them later). Then proceed to the next step (*Customize target index*).
- Change the **Index name** to **margies-index**.
- Ensure that the **Key** is set to **metadata_storage_path** and leave the **Suggester name** blank and **Search mode** at its default.

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

i We provided a default index for you. You can delete the fields you don't need. Everything is editable, but once the index is created, deleting or changing existing fields will require re-indexing your documents.

Index name * ⓘ
 margies-index ✓

Key * ⓘ
 metadata_storage_path

Suggester name
 Search mode ⓘ
 analyzingInfixMatching

- Make the following changes to the index fields, leaving all other fields with their default settings

(**IMPORTANT:** you may need to scroll to the right to see the entire table):

Field Name	Retrievable	Filterable	Sortable	Facetable	Searchable
metadata_storage_size	✓	✓	✓	✓	✓
metadata_storage_last_modified	✓	✓	✓	✓	✓
metadata_storage_name	✓	✓	✓	✓	✓
metadata_author	✓	✓	✓	✓	✓
locations	✓	✓	✓	✓	✓
keyphrases	✓	✓	✓	✓	✓
language	✓	✓	✓	✓	✓

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable	Analyzer	Suggester
content	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
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 checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			...
metadata_storage_last_modified	Edm.DateTi...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>			...
metadata_storage_content_md5	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_name	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
metadata_storage_path	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_storage_file_extension	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_author	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
metadata_title	Edm.String	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		...
metadata_creation_date	Edm.DateTi...	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			...
locations	Collection(E...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
keyphrases	Collection(E...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
language	Edm.String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
merged_content	Edm.String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...
text	Collection(E...	<input checked="" type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standard - Luce...	...

Previous: Add cognitive skills (Optional) Next: Create an indexer

- Double-check your index field selections, paying particular attention to ensure that the correct **Retrievable**, **Filterable**, **Sortable**, **Facetable**, and **Searchable** options are selected for each field (it can be difficult to change them later). Then proceed to the next step (*Create an indexer*).
- Change the **Indexer name** to **margies-indexer**.
- Leave the **Schedule** set to **Once**.
- Expand the **Advanced options**, and ensure that the **Base-64 encode keys** option is selected (generally **encoding keys make the index more efficient**).

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

Indexer

Name *

Schedule ☒ Once ☐ Hourly ☐ Daily ☐ Custom

Description

Advanced options

Base-64 Encode Keys ☒

Max failed items

Max failed items per batch

Batch size

Excluded extensions

Indexed extensions

Previous: Customize target index Submit

14. Select **Submit** to create the data source, skillset, index, and indexer. The indexer is run automatically and runs the indexing pipeline, which:
 1. Extracts the document metadata fields and content from the data source
 2. Runs the skillset of cognitive skills to generate additional enriched fields
 3. Maps the extracted fields to the index.
15. In the bottom half of the **Overview** page for your Azure AI Search resource, view the **Indexers** tab, which should show the newly created **margies-indexer**. Wait a few minutes, and click ↻ **Refresh** until the **Status** indicates success.

Search the index

Now that you have an index, you can search it.

1. At the top of the **Overview** page for your Azure AI Search resource, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter `*` (a single asterisk), and then select **Search**.
This query retrieves all documents in the index in JSON format. Examine the results and note the fields for each document, which contain document content, metadata, and enriched data extracted by the cognitive skills you selected.
3. In the **View** menu, select **JSON view** and note that the JSON request for the search is shown, like this:

```
{"search": "*" }
```
4. Modify the JSON request to include the **count** parameter as shown here:

```
{"search": "*", "count": true }
```
5. Submit the modified search. This time, the results include a **@odata.count** field at the top of the results that indicates the number of documents returned by the search.
6. Try the following query:

```
{"search": "count": true, "select": "metadata_storage_name,metadata_author,locations" }
```


This time the results include only the file name, author, and any locations mentioned in the document content. The file name and author are in the **metadata_storage_name** and **metadata_author** fields, which were extracted from the source document. The **locations** field was generated by a cognitive skill.
7. Now try the following query string:

```
{"search": "New York", "count": true, "select": "metadata_storage_name,keyphrases" }
```


This search finds documents that mention "New York" in any of the searchable fields, and returns the file name and key phrases in the document.
8. Let's try one more query:

```
{"search": "New York", "count": true, "select": "metadata_storage_name", "filter": "metadata_author eq 'Reviewer'" }
```

This query returns the filename of any documents authored by *Reviewer* that mention "New York".

Explore and modify definitions of search components

The components of the search solution are based on JSON definitions, which you can view and edit in the Azure portal.

While you can use the portal to create and modify search solutions, it's often desirable to define the search objects in JSON and use the Azure AI Service REST interface to create and modify them.

Get the endpoint and key for your Azure AI Search resource

1. In the Azure portal, return to the **Overview** page for your Azure AI Search resource; and in the top section of the page, find the **Url** for your resource (which looks like https://resource_name.search.windows.net) and copy it to the clipboard.
2. In Visual Studio Code, in the Explorer pane, expand the **01-azure-search** folder and its **modify-search** subfolder, and select **modify-search.cmd** to open it. You will use this script file to run *cURL* commands that submit JSON to the Azure AI Service REST interface.
3. In **modify-search.cmd**, replace the **YOUR_SEARCH_URL** placeholder with the URL you copied to the clipboard.
4. In the Azure portal, view the **Keys** page for your Azure AI Search resource, and copy the **Primary admin key** to the clipboard.
5. In Visual Studio Code, replace the **YOUR_ADMIN_KEY** placeholder with the key you copied to the clipboard.
6. Save the changes to **modify-search.cmd** (but don't run it yet!)

Review and modify the skillset

1. In Visual studio Code, in the **modify-search** folder, open **skillset.json** ([skillset](#)). This shows a JSON definition for **margies-skillset**.
2. At the top of the skillset definition, note the **cognitiveServices** object, which is used to connect your Azure AI Services resource to the skillset.
3. In the Azure portal, open your Azure AI Services resource (not your Azure AI Search resource!) and view its **Keys** page. Then copy **Key 1** to the clipboard.
4. In Visual Studio Code, in **skillset.json**, replace the **YOUR_COGNITIVE_SERVICES_KEY** placeholder with the Azure AI Services key you copied to the clipboard.
5. Scroll through the JSON file, noting that it includes definitions for the skills you created using the Azure AI Search user interface in the Azure portal. At the bottom of the list of skills, an additional skill has been added with the following definition:

```
{"@odata.type": "#Microsoft.Skills.Text.V3.SentimentSkill", "defaultLanguageCode": "en",
"name": "get-sentiment", "description": "New skill to evaluate sentiment", "context":
"/document", "inputs": [{"name": "text", "source": "/document/merged_content" },
{"name": "languageCode", "source": "/document/language"}], "outputs": [{"name":
"sentiment", "targetName": "sentimentLabel"}]}
```

The new skill is named **get-sentiment**, and for each **document** level in a document, it, will evaluate the text found in the **merged_content** field of the document being indexed (which includes the source content as well as any text extracted from images in the content). It uses the extracted **language** of the document (with a default of English), and evaluates a label for the sentiment of the content. Values for the sentiment label can be **"positive"**, **"negative"**, **"neutral"**, or **"mixed"**. This label is then output as a new field named **sentimentLabel**.

6. Save the changes you've made to **skillset.json**.

Review and modify the index

1. In Visual studio Code, in the **modify-search** folder, open **index.json** ([index](#)). This shows a JSON definition for **margies-index**.
2. Scroll through the index and view the field definitions. Some fields are based on metadata and content in the source document, and others are the results of skills in the skillset.
3. At the end of the list of fields that you defined in the Azure portal, note that two additional fields have been added:

```
{ "name": "sentiment", "type": "Edm.String", "facetable": false, "filterable": true,
  "retrievable": true, "sortable": true }, { "name": "url", "type": "Edm.String",
  "facetable": false, "filterable": true, "retrievable": true, "searchable": false,
  "sortable": false }
```

4. The **sentiment** field will be used to add the output from the **get-sentiment** skill that was added the skillset. The **url** field will be used to add the URL for each indexed document to the index, based on the **metadata_storage_path** value extracted from the data source. Note that index already includes the **metadata_storage_path** field, but it's used as the index key and Base-64 encoded, making it efficient as a key but requiring client applications to decode it if they want to use the actual URL value as a field. Adding a second field for the unencoded value resolves this problem.

Review and modify the indexer

1. In Visual studio Code, in the **modify-search** folder, open **indexer.json** ([indexer](#)). This shows a JSON definition for **margies-indexer**, which maps fields extracted from document content and metadata (in the **fieldMappings** section), and values extracted by skills in the skillset (in the **outputFieldMappings** section), to fields in the index.
2. In the **fieldMappings** list, note the mapping for the **metadata_storage_path** value to the base-64 encoded key field (i.e., "mappingFunction": { "name": "base64Encode" }). This was created when you assigned the **metadata_storage_path** as the key and selected the option to encode the key in the Azure portal. Additionally, a new mapping explicitly maps the same value to the **url** field, but without the Base-64 encoding:

```
{ "sourceFieldName" : "metadata_storage_path", "targetFieldName" : "url" }
```

All of the other metadata and content fields in the source document are implicitly mapped to fields of the same name in the index.

3. Review the **outputFieldMappings** section, which maps outputs from the skills in the skillset to index fields. Most of these reflect the choices you made in the user interface, but the following mapping has been added to map the **sentimentLabel** value extracted by your sentiment skill to the **sentiment** field you added to the index:

```
{ "sourceFieldName": "/document/sentimentLabel", "targetFieldName": "sentiment" }
```

Use the REST API to update the search solution

1. Right-click the **modify-search** folder and open an integrated terminal.
2. In the terminal pane for the **modify-search** folder, enter the following command to run the **modify-search.cmd** script ([modify-search.cmd](#)), which submits the JSON definitions to the REST interface and initiates the indexing.

```
./modify-search
```

3. When the script has finished, return to the **Overview** page for your Azure AI Search resource in the Azure portal and view the **Indexers** page. The periodically select **Refresh** to track the progress of the indexing operation. It may take a minute or so to complete.

There may be some warnings for a few documents that are too large to evaluate sentiment. Often sentiment analysis is performed at the page or sentence level rather than the full document; but in this case scenario, most of the documents - particularly the hotel reviews, are short enough for useful document-level sentiment scores to be evaluated.

Query the modified index

1. At the top of the blade for your Azure AI Search resource, select **Search explorer**.

2. In Search explorer, in the **Query string** box, submit the following JSON query:

```
{ "search": "London", "select": "url,sentiment,keyphrases", "filter": "metadata_author eq 'Reviewer' and sentiment eq 'positive'" }
```

This query retrieves the **url**, **sentiment**, and **keyphrases** for all documents that mention *London* authored by *Reviewer* that have a positive **sentiment** label (in other words, positive reviews that mention London)

3. Close the **Search explorer** page to return to the **Overview** page.

Create a search client application

Now that you have a useful index, you can use it from a client application. You can do this by consuming the REST interface, submitting requests and receiving responses in JSON format over HTTP; or you can use the software development kit (SDK) for your preferred programming language. In this exercise, we'll use the SDK.

Note: You can choose to use the SDK for either **C#** or **Python**. In the steps below, perform the actions appropriate for your preferred language.

Get the endpoint and keys for your search resource

1. In the Azure portal, on the **Overview** page for your Azure AI Search resource, note the **Url** value, which should be similar to https://_your_resource_name_.search.windows.net. This is the endpoint for your search resource.
2. On the **Keys** page, note that there are two **admin** keys, and a single **query** key. An *admin* key is used to create and manage search resources; a *query* key is used by client applications that only need to perform search queries.

You will need the endpoint and query key for your client application.

Prepare to use the Azure AI Search SDK

1. In Visual Studio Code, in the **Explorer** pane, browse to the **01-azure-search** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **margies-travel** folder and open an integrated terminal. Then install the Azure AI Search SDK package by running the appropriate command for your language preference:

C#

```
dotnet add package Azure.Search.Documents --version 11.1.1
```

Python

```
pip install azure-search-documents==11.0.0
```

3. View the contents of the **margies-travel** folder, and note that it contains a file for configuration settings:
 - **C#:** appsettings.json
 - **Python:** .env

Open the configuration file and update the configuration values it contains to reflect the **endpoint** and **query key** for your Azure AI Search resource. Save your changes.

Explore code to search an index

The **margies-travel** folder contains code files for a web application (a Microsoft C# *ASP.NET Razor* web application or a Python *Flask* application), which includes search functionality.

1. Open the following code file in the web application, depending on your choice of programming language:
 - **C#**: Pages/Index.cshtml.cs
 - **Python**: app.py (View here: [app](#))
2. Near the top of the code file, find the comment **Import search namespaces**, and note the namespaces that have been imported to work with the Azure AI Search SDK:
3. In the **search_query** function, find the comment **Create a search client**, and note that the code creates a **SearchClient** object using the endpoint and query key for your Azure AI Search resource

```
def search_query(search_text, filter_by=None, sort_order=None):
    try:
        # Create a search client
        azure_credential = AzureKeyCredential(search_key)
        search_client = SearchClient(search_endpoint, search_index, azure_credential)
        # Submit search query
        results = search_client.search(search_text,
                                       search_mode="all",
                                       include_total_count=True,
                                       filter=filter_by,
                                       order_by=sort_order,
                                       facets=['metadata_author'],
                                       highlight_fields='merged_content-3,imageCaption-3',

                                       select =
url,metadata_storage_name,metadata_author,metadata_storage_size,metadata_storage_last_modified,language,sentiment,merged_content,keyphrases,locations,imageTags,imageCaption")
        return results
    except Exception as ex:
        raise ex
```

4. In the **search_query** function, find the comment **Submit search query**, and review the code to submit a search for the specified text with the following options:
 - A *search mode* that requires **all** of the individual words in the search text are found.
 - The total number of documents found by the search is included in the results.
 - The results are filtered to include only documents that match the provided filter expression.
 - The results are sorted into the specified sort order.
 - Each discrete value of the **metadata_author** field is returned as a *facet* that can be used to display pre-defined values for filtering.
 - Up to three extracts of the **merged_content** and **imageCaption** fields with the search terms highlighted are included in the results.
 - The results include only the fields specified.

Explore code to render search results

The web app already includes code to process and render the search results.

1. Open the following code file in the web application, depending on your choice of programming language:
 - **C#**: Pages/Index.cshtml
 - **Python**: templates/search.html (View here: [search.html](#))
2. Examine the code, which renders the page on which the search results are displayed. Observe that:
 - The page begins with a search form that the user can use to submit a new search (in the Python version of the application, this form is defined in the **base.html** template), which is referenced at the beginning of the page.
 - A second form is then rendered, enabling the user to refine the search results. The code for this form:
 - Retrieves and displays the count of documents from the search results.
 - Retrieves the facet values for the **metadata_author** field and displays them as an option list for filtering.
 - Creates a drop-down list of sort options for the results.
 - The code then iterates through the search results, rendering each result as follows:
 - Display the **metadata_storage_name** (file name) field as a link to the address in the **url** field.
 - Displaying *highlights* for search terms found in the **merged_content** and **imageCaption** fields to help show the search terms in context.
 - Display the **metadata_author**, **metadata_storage_size**, **metadata_storage_last_modified**, and **language** fields.
 - Display the **sentiment** label for the document. Can be positive, negative, neutral, or mixed.
 - Display the first five **keyphrases** (if any).
 - Display the first five **locations** (if any).
 - Display the first five **imageTags** (if any).

Run the web app

1. Return to the integrated terminal for the **margies-travel** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
flask run
```

2. In the message that is displayed when the app starts successfully, follow the link to the running web application (<http://localhost:5000/> or <http://127.0.0.1:5000/>) to open the Margies Travel site in a web browser.
3. In the Margie's Travel website, enter **London hotel** into the search box and click **Search**.
4. Review the search results. They include the file name (with a hyperlink to the file URL), an extract of the file content with the search terms (*London* and *hotel*) emphasized, and other attributes of the file from the index fields.

5. Observe that the results page includes some user interface elements that enable you to refine the results. These include:
 - A *filter* based on a facet value for the **metadata_author** field. This demonstrates how you can use *facetable* fields to return a list of *facets* - **fields with a small set of discrete values that can be displayed as potential filter values in the user interface**.
 - The ability to *order* the results based on a specified field and sort direction (ascending or descending). The default order is based on *relevancy*, which is calculated as a **search.score()** value based on a *scoring profile* that evaluates the frequency and importance of search terms in the index fields.
6. Select the **Reviewer** filter and the **Positive to negative** sort option, and then select **Refine Results**.
7. Observe that the results are filtered to include only reviews, and sorted based on the sentiment label.
8. In the **Search** box, enter a new search for **quiet hotel in New York** and review the results.
9. Try the following search terms:
 - **Tower of London** (observe that this term is identified as a *key phrase* in some documents).
 - **skyscraper** (observe that this word doesn't appear in the actual content of any documents, but is found in the *image captions* and *image tags* that were generated for images in some documents).
 - **Mojave desert** (observe that this term is identified as a *location* in some documents).
10. Close the browser tab containing the Margie's Travel web site and return to Visual Studio Code. Then in the Python terminal for the **margies-travel** folder (where the dotnet or flask application is running), enter Ctrl+C to stop the app.

More information

To learn more about Azure AI Search, see the [Azure AI Search documentation](#).

Knowledge Check

1. You want to find information in Microsoft Word documents that are stored in an Azure Storage blob container. What should you do to ensure the files can be accessed by Azure AI Search? *

- ☐ Add a JSON file that defines an Azure AI Search index to the blob container
- ☐ Enable anonymous access for the blob container
- ☒ In an Azure AI Services resource, add a data source that references the container where the files are stored

✓ Correct. To search files in a blob container, you should create a data source

2. You are creating an index that includes a field named `modified_date`. You want to ensure that the `modified_date` field can be included in search results. Which attribute must you apply to the `modified_date` field in the index definition? *

- ☐ searchable
- ☐ filterable
- ☒ retrievable

✓ Correct. To enable a field to be included in the results, you must make it retrievable.

3. You have created a data source and an index. What must you create to map the data values in the data source to the fields in the index? *

- ☐ A synonym map
- ☒ An indexer

✓ Correct. Use an indexer to map data to index fields.

- ☐ A suggester

4. You want to create a search solution that uses a built-in AI skill to determine the language in which each indexed document is written, and enrich the index with a field indicating the language. Which kind of Azure AI Search object must you create? *

- ☐ Synonym map
- ☒ Skillset

✓ Correct. A skillset enables you to define an enrichment pipeline composed of AI skills.

- ☐ Scoring Profile

5. You want your search solution to show results in descending order of the `file_size` field value. What is the simplest way to accomplish this goal? *

- ☐ Create a scoring profile that boosts results based on the `file_size` field
- ☐ Make the `file_size` field facetable, and include a facet parameter that specifies the `file_size` field in queries.
- ☒ Make the `file_size` field sortable, and include an `orderby` parameter that specifies the `file_size` field in queries.

✓ Correct. Making a field sortable enables you to apply an `orderby` parameter to sort results by that field.

6. You have created a search solution. Users want to be able to enter a partial search expression and have the user interface automatically complete the input. What should you add to the index? *

- ☒ A suggester

✓ Correct. A suggester makes it possible to implement autocomplete and suggestions.

- ☐ A synonym map.
- ☐ A scoring profile.

Summary

In this module, you learned how to use Azure AI Search to create an AI Search solution that consists of:

- A *data source* where the data to be indexed is stored (though you can also push data directly into an index by using the API).
- A *skillset* that defines an enrichment pipeline of **cognitive skills to enrich** the index data.
- An *index* that defines fields, which the user can query.
- An *indexer* that **populates the fields in the index** with values extracted from the source data.

Now that you've completed this module, you've learned to:

- Create an Azure AI Search solution
- Develop a search application

You can use the Azure AI Search REST APIs or SDKs to create and manage index objects, and to implement a client application that queries the index to retrieve information.

For more information about Azure AI Search, take a look at the [service documentation](#).