

## 4.7 - Search data outside the Azure platform in Azure AI Search using Azure Data Factory

- [Overview](#)
  - [Learning objectives](#)
  - [Introduction](#)
  - [Index data from external data sources using Azure Data Factory](#)
    - [Push data into a search index using Azure Data Factory \(ADF\)](#)
      - [Create an ADF pipeline to push data into a search index](#)
        - [Create a search index](#)
        - [Create a pipeline using the ADF Copy Data tool](#)
        - [Create the source linked service](#)
        - [Create the target linked service](#)
        - [Map source fields to target fields](#)
        - [Run the pipeline to push the data into the index](#)
      - [Limitations of using the built-in Azure AI Search as a linked service](#)
  - [Index any data using the Azure AI Search push API](#)
    - [Supported REST API operations](#)
    - [How to call the search REST API](#)
      - [Add data to an index](#)
    - [Use .NET Core to index any data](#)
      - [Work out your optimal batch size](#)
      - [Implement an exponential backoff retry strategy](#)
      - [Use threading to improve performance](#)
  - [Exercise: Add to an index using the push API](#)
    - [Set up your Azure resources](#)
    - [Copy Azure AI Search service REST API information](#)
    - [Download example code](#)
    - [Edit the code to implement threading and a backoff and retry strategy](#)
    - [Delete exercise resources](#)
  - [Knowledge Check](#)
  - [Summary](#)
- 

### Overview

Use Azure Data Factory to add data that resides inside or outside the Azure platform into your search indexes.

### Learning objectives

In this module, you'll learn how to:

- Use Azure Data Factory to copy data into an Azure AI Search Index

- Use the Azure AI Search push API to add to an index from any external data source
- 

## Introduction

Azure AI Search supports two main ways to get data into a search index. So far, you've seen how to *pull* data using indexers. The most flexible way to get data into an index is to *push* the data into it.

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

- Use Azure Data Factory to copy data into an Azure AI Search Index.
  - Use the Azure AI Search push API to add to an index from any external data source.
- 

## Index data from external data sources using Azure Data Factory

**Adding external data that doesn't reside in Azure** is a common need in an organization's search solution. Azure AI Search is flexible as it allows many ways to create and push data into indexes.

### Push data into a search index using Azure Data Factory (ADF)

A first approach is a zero-code option for pushing data into an index using ADF. ADF comes with connections to nearly 100 different data stores. With connectors like HTTP and REST that allow you to connect an unlimited number of data stores.

These data stores are used as a source or a target (called **sinks** in the copy activity) in pipelines. The Azure AI Search index connector can be used as a sink in a copy activity.

### Create an ADF pipeline to push data into a search index

The steps you need to take to use an ADF pipeline to push data into a search index are:

1. Create an Azure AI Search index with all the fields you want to store data in.
2. Create a pipeline with a **copy data step**.
3. Create a data source connection to where your data resides.
4. Create a **sink to connect to your search index**.
5. Map the fields from your source data to your search index.
6. Run the pipeline to push the data into the index.

For example, imagine you've customer data in JSON format that is hosted externally. You want to copy these customers into a search index. The JSON is in this format:

```
{
  "_id": "5fed1b38309495de1bc4f653",
  "firstName": "Sims",
  "lastName": "Arnold",
  "isAlive": false,
  "age": 35,
  "address": {
    "streetAddress": "Sumner Place",
```

```

    "city": "Canoochee",
    "state": "Palau",
    "postalCode": 1558
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "+1 (830) 465-2965"
    },
    {
      "type": "home",
      "number": "+1 (889) 439-3632"
    }
  ]
}

```

## Create a search index


Create an Azure AI Search service and an index to store this information in. If you've completed the [Create an Azure AI Search solution](#) module, then you've seen how to do this. Follow the steps to create the search service but stop at the point of importing data. **As pushing data into an index doesn't need you to create an indexer or skillset.**

Create an index and add these fields and properties:

Search explorer **Fields** CORS Scoring profiles Semantic configurations

+ Add field + Add subfield Delete Autocomplete settings

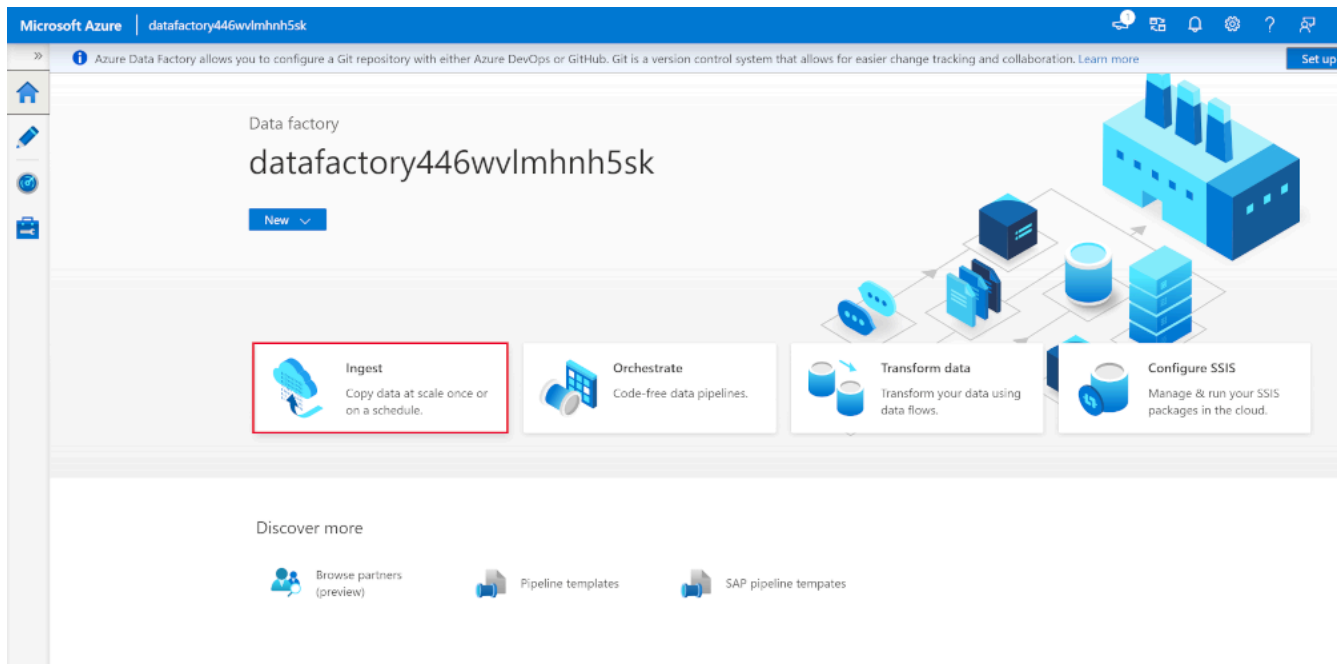
Search field names

Field name	Type	Retrievable	Filterable	Sortable	Facetable	Searchable	Analyzer	Dimensions
		<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
 id	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
firstName	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
lastName	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
isAlive	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
streetAddress	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
city	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
state	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
postalCode	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
phoneNumber_type	String	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
phoneNumber	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	
age	String	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Standa... ▼	

At the moment **you have to create the index first, as ADF can't create indexes.**

## Create a pipeline using the ADF Copy Data tool

Open the [Azure Data Factory Studio](#) and select your Azure subscription and data factory name.



1. Select **Ingest**.
2. Select **Next**.

Note: You can choose to schedule the pipeline if your data is changing and you need to keep your index up-to-date. For this example, you'll import the data once.

## Create the source linked service

1. In **Source type**, select **HTTP**.
2. Next to **Connection**, select **+ New connection**.

3. In the **New connection** pane, in **Name** enter **dataLocation**.

4. In the **Base URL**, enter where your JSON file resides, in this example enter <https://raw.githubusercontent.com/Azure-Samples/azure-sql-db-import-data/main/json/user1.json>.
5. In **Authentication type**, select **Anonymous**.
6. Select **Create**.
7. Select **Next**.

### Copy Data tool

✓ Properties

2 Source

Dataset

Configuration

3 Destination

4 Settings

5 Review and finish

#### File format settings

File format  
JSON ▼ Preview data

☐ Export as-is to JSON files or Azure Cosmos DB collection

Compression type  
▼

Encoding ⓘ  
Default(UTF-8) ▼

Additional columns ⓘ  
+ New

< Previous Next >

8. In **File format**, select **JSON**.
9. Select **Next**.

## Create the target linked service

1. In **Destination type**, select **Azure Search**. Then select **+ New connection**.

The screenshot shows the 'Copy Data tool' configuration interface. On the left, a sidebar lists steps: Properties, Source, Destination, Dataset, Configuration, Settings, and Review and finish. The 'Destination' step is selected. The main area is titled 'Destination data store' and contains a 'Destination type' dropdown set to 'Azure Search' and a 'Connection' dropdown set to 'Select...' with a '+ New connection' link. To the right, the 'New connection' pane is open, showing the following fields: 'Name' (search\_index), 'Description' (empty), 'Connect via integration runtime' (AutoResolveIntegrationRuntime), 'Account selection method' (From Azure subscription), 'Azure subscription' (AI Subscription), and 'Service name' (azcsearchtest). The 'Create' button is visible at the bottom of the 'New connection' pane.

2. In the **New connection** pane, in **Name** enter **search\_index**.
3. In **Azure subscription**, select your Azure subscription.
4. In **Service name**, select your Azure AI Search service.
5. Select **Create**.
6. On the **Destination data store** pane, in **Target**, select the search index you created.

## Map source fields to target fields

1. Select **Next**.

## Copy Data tool

Properties

Source

Destination

Dataset

Configuration

Settings

Review and finish

### Schema mapping

Choose how source and destination columns are mapped

Source

Target

customerinfo

Error

Please fix the errors

Table mappings (1)

New mapping

Clear

Delete

Advanced editor

Collection reference

Map complex values to string

Name	Type	Collection reference	Column name	Inc
_id	abc string	→	Edit column	✓
firstName	abc string	→	firstName (String)	✓
lastName	abc string	→	lastName (String)	✓
isAlive	% boolean	→	isAlive (String)	✓
age	123 integer	→	age (String)	✓
address	any object			
streetAddress	abc string	→	streetAddress (String)	✓
city	abc string	→	city (String)	✓
state	abc string	→	state (String)	✓
postalCode	123 integer	→	postalCode (String)	✓
phoneNumbers[0]	any any	→	Edit column	✓
phoneNumbers[0]	any any	→	Edit column	✓

Azure Search sink properties

Advanced

Previous

Next

Cancel

- If you created an index with field names that match the JSON attributes, ADF will automatically map the JSON to the field in your search index.
- In the above example, three fields in the JSON document need mapping to fields in the index.
- Map your fields, then select **Next**.
- On the **Settings** pane, in **Task name**, enter **jsonToSearchIndex**.
- Select **Next**.

## Run the pipeline to push the data into the index

- On the **Summary** pane, select **Next**.

## Copy Data tool

- ✓ Properties
- ✓ Source
- ✓ Destination
- ✓ Settings
- 5 Review and finish
- Review
- Deployment



HTTP



Azure Search

### Deployment complete

Deployment step	Status
Validating copy runtime environment	✓ Succeeded
> Creating datasets	✓ Succeeded
> Creating pipelines	✓ Succeeded
> Running pipelines	✓ Succeeded

Datasets and pipelines have been created. You can now monitor and edit the copy pipelines or click finish to close Copy Data Tool.



Finish

Edit pipeline

Monitor

2. Once the pipeline has been validated and deployed, select **Finish**.

The pipeline has been deployed and run. **The JSON document will have been added to your search index.** You can use the Azure portal and run a search in the search explorer. You should see the imported JSON data.



## Search explorer ...

azcsearchtest

Index

customerinfo

View

API version

2023-07-01-Preview

Search

Query string

sims

Request URL

https://azcsearchtest.search.windows.net/indexes/customerinfo/docs?api-version=2023-07-01-Preview&amp;search=sims

Results

```
1 {
2   "@odata.context": "https://azcsearchtest.search.windows.net/indexes('customerinfo')/$metadata#docs(*)",
3   "value": [
4     {
5       "@search.score": 0.2876821,
6       "id": "5fed1b38309495de1bc4f653",
7       "firstName": "Sims",
8       "lastName": "Arnold",
9       "isAlive": "False",
10      "streetAddress": "Sumner Place",
11      "city": "Canoochee",
12      "state": "Palau",
13      "postalCode": "1558",
14      "phoneNumber_type": "home",
15      "phoneNumber": "+1 (830) 465-2965",
16      "age": "35"
17    }
18  ]
19 }
```

Following these steps you've seen how you can **push data into an index**. The pipeline you've created by default merges updates into the index. If you amended the JSON data and rerun the pipeline, the search index would be updated. You can change the write behavior to upload only if you want the data to be replaced each time you run your pipeline.

## Limitations of using the built-in Azure AI Search as a linked service

At the moment, the Azure AI Search linked service as a sink only supports these fields:

Azure AI Search data type
String
Int32
Int64
Double
Boolean
DateTimeOffset

This means **ComplexTypes and arrays aren't currently supported**. Looking at the JSON document above this means that it isn't possible to map all the phone numbers for the customer. Only the first telephone number has been mapped.

---

# Index any data using the Azure AI Search push API

The REST API is the **most flexible way to push data into an Azure AI Search index**. You can use any programming language or interactively with any app that can post JSON requests to an endpoint.

Here, you'll see how to use the REST API effectively and explore the available operations. Then you'll look at .NET Core code and see how to optimize adding large amounts of data through the API.

## Supported REST API operations

There are two supported REST APIs provided by AI Search: Search and management APIs. This module focuses on the **search REST APIs** that provide operations on five features of search:

Feature	Operations
Index	Create, delete, update, and configure.
Document	Get, add, update, and delete.
Indexer	Configure data sources and scheduling on limited data sources.
Skillset	Get, create, delete, list, and update.
Synonym map	Get, create, delete, list, and update.

## How to call the search REST API

If you want to call any of the search APIs you need to:

- Use the HTTPS endpoint (over the default port 443) provided by your search service, you must include an **api-version** in the URI.
- The request header must include an **api-key** attribute.

To find the endpoint, api-version, and api-key go to the Azure portal.

# Search explorer

azcsearchtest

Index

customerinfo

View

API version

2023-07-01-Preview

Search

Query string

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

Request URL

https://azcsearchtest.search.windows.net/indexes/customerinfo/docs?api-version=2023-07-01-Preview&search=\*

Results

In the portal, navigate to your search service, then select **Search explorer**. The REST API endpoint is in the **Request URL** field. The first part of the URL is the endpoint (for example <https://azcsearchtest.search.windows.net>), and the query string shows the `api-version` (for example `api-version=2023-07-01-Preview`).

## Search management

 Indexes

 Indexers

 Data sources

 Aliases

 Skillsets

 Debug sessions

## Settings

 Semantic search (Preview)

 Knowledge Center

 **Keys**
 Scale

 Search traffic analytics

## API access control



☒ API keys

☐ Role-based access control ⓘ



☐ Both

## Manage admin keys


Primary admin key

   Regenerate

Secondary admin key

   Regenerate

## Manage query keys

 Add  Delete

To find the `api-key` on the left, select **Keys**. The primary or secondary admin key can be used if you're using the REST API to do more than just querying the index.

If all you need is to search an index, you can create and use query keys.

**To add, update, or delete data in an index you need to use an admin key.**

## Add data to an index

Use an HTTP POST request using the **indexes feature** in this format:

```
POST https://[service name].search.windows.net/indexes/[index name]/docs/index?api-version=[api-version]
```

The **body of your request needs to let the REST endpoint know the action** to take on the document, which document to apply the action too, and what data to use.

The JSON must be in this format:

```
{
  "value": [
    {
      "@search.action": "upload (default) | merge | mergeOrUpload | delete",
      "key_field_name": "unique_key_of_document", (key/value pair for key field from
index schema)
      "field_name": field_value (key/value pairs matching index schema)
      ...
    },
    ...
  ]
}
```

```
]
}
```

Action	Description
<b>upload</b>	Similar to an upsert in SQL, the document will be created or replaced.
<b>merge</b>	Merge updates an existing document with the specified fields. Merge will fail if no document can be found.
<b>mergeOrUpload</b>	Merge updates an existing document with the specified fields, and uploads it if the document doesn't exist.
<b>delete</b>	Deletes the whole document, you only need to specify the key_field_name.

If your request is successful, the API will return a **200 status code**.

Note: For a full list of all the response codes and error messages, see [Add, Update or Delete Documents \(Azure AI Search REST API\)](#)

This example JSON uploads the customer record in the previous unit:

```
{
  "value": [
    {
      "@search.action": "upload",
      "id": "5fed1b38309495de1bc4f653",
      "firstName": "Sims",
      "lastName": "Arnold",
      "isAlive": false,
      "age": 35,
      "address": {
        "streetAddress": "Sumner Place",
        "city": "Canoochee",
        "state": "Palau",
        "postalCode": "1558"
      },
      "phoneNumbers": [
        {
          "phoneNumber": {
            "type": "home",
            "number": "+1 (830) 465-2965"
          }
        },
        {
          "phoneNumber": {
            "type": "home",
            "number": "+1 (889) 439-3632"
          }
        }
      ]
    }
  ]
}
```

You can add as many documents in the value array as you want. However, for optimal performance consider batching the documents in your requests **up to a maximum of 1,000 documents, or 16 MB in total size**.

## Use .NET Core to index any data

For best performance use the latest `Azure.Search.Document` client library, currently version 11. You can install the client library with NuGet:

```
dotnet add package Azure.Search.Documents --version 11.4.0
```

How your index performs is based on six key factors:

- The search service tier and **how many replicas and partitions** you've enabled.
- The **complexity** of the index schema. **Reduce how many properties (searchable, facetable, sortable) each field has.**
- The **number of documents** in each batch, the best size will depend on the index schema and the size of documents.
- How multithreaded your approach is.
- Handling errors and throttling. Use an exponential backoff retry strategy.
- **Where your data resides, try to index your data as close to your search index. For example, run uploads from inside the Azure environment.**

## Work out your optimal batch size

As working out the best batch size is a key factor to improve performance, let's look at an approach in code.

```
public static async Task TestBatchSizesAsync(SearchClient searchClient, int min = 100,
int max = 1000, int step = 100, int numTries = 3)
{
    DataGenerator dg = new DataGenerator();

    Console.WriteLine("Batch Size \t Size in MB \t MB / Doc \t Time (ms) \t MB /
Second");
    for (int numDocs = min; numDocs <= max; numDocs += step)
    {
        List<TimeSpan> durations = new List<TimeSpan>();
        double sizeInMb = 0.0;
        for (int x = 0; x < numTries; x++)
        {
            List<Hotel> hotels = dg.GetHotels(numDocs, "large");

            DateTime startTime = DateTime.Now;
            await UploadDocumentsAsync(searchClient, hotels).ConfigureAwait(false);
            DateTime endTime = DateTime.Now;
            durations.Add(endTime - startTime);

            sizeInMb = EstimateObjectSize(hotels);
        }

        var avgDuration = durations.Average(timeSpan => timeSpan.TotalMilliseconds);
        var avgDurationInSeconds = avgDuration / 1000;
```

```

        var mbPerSecond = sizeInMb / avgDurationInSeconds;

        Console.WriteLine("{0} \t\t {1} \t\t {2} \t\t {3} \t {4}", numDocs,
Math.Round(sizeInMb, 3), Math.Round(sizeInMb / numDocs, 3), Math.Round(avgDuration, 3),
Math.Round(mbPerSecond, 3));

        // Pausing 2 seconds to let the search service catch its breath
        Thread.Sleep(2000);
    }

    Console.WriteLine();
}

```

The approach is to **increase the batch size and monitor the time it takes to receive a valid response**. The code loops from 100 to 1000, in 100 document steps. For each batch size, it outputs the document size, time to get a response, and the average time per MB. Running this code gives results like this:

```

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...

```

In the above example, the best batch size for throughput is **2.499** MB per second, **800** documents per batch.

## Implement an exponential backoff retry strategy

If your index starts to **throttle requests due to overloads**, it responds with a **503 (request rejected due to heavy load)** or **207 (some documents failed in the batch)** status.

You have to handle these responses and a good strategy is to **backoff**. **Backing off means pausing for some time before retrying** your request again. If you increase this time for each error, you'll be exponentially backing off.

Look at this code:

```

// Implement exponential backoff
do
{
    try
    {

```

```

    attempts++;
    result = await searchClient.IndexDocumentsAsync(batch).ConfigureAwait(false);

    var failedDocuments = result.Results.Where(r => r.Succeeded != true).ToList();

    // handle partial failure
    if (failedDocuments.Count > 0)
    {
        if (attempts == maxRetryAttempts)
        {
            Console.WriteLine("[MAX RETRIES HIT] - Giving up on the batch starting
at {0}", id);
            break;
        }
        else
        {
            Console.WriteLine("[Batch starting at doc {0} had partial failure]",
id);
            Console.WriteLine("[Retrying {0} failed documents] \n",
failedDocuments.Count);

            // creating a batch of failed documents to retry
            var failedDocumentKeys = failedDocuments.Select(doc =>
doc.Key).ToList();
            hotels = hotels.Where(h =>
failedDocumentKeys.Contains(h.HotelId)).ToList();
            batch = IndexDocumentsBatch.Upload(hotels);

            Task.Delay(delay).Wait();
            delay = delay * 2;
            continue;
        }
    }

    return result;
}
catch (RequestFailedException ex)
{
    Console.WriteLine("[Batch starting at doc {0} failed]", id);
    Console.WriteLine("[Retrying entire batch] \n");

    if (attempts == maxRetryAttempts)
    {
        Console.WriteLine("[MAX RETRIES HIT] - Giving up on the batch starting at
{0}", id);
        break;
    }

    Task.Delay(delay).Wait();
    delay = delay * 2;
}
} while (true);

```

The code keeps track of failed documents in a batch. If an error happens, it waits for a delay and then doubles the delay for the next error.



Finally, there's a maximum number of retries, and if this maximum number is reached the program exists.

## Use threading to improve performance

You can complete your document uploading app by combining the above backoff strategy with a **threading approach**. Here's some example code:

```
public static async Task IndexDataAsync(SearchClient searchClient, List<Hotel>
hotels, int batchSize, int numThreads)
{
    int numDocs = hotels.Count;
    Console.WriteLine("Uploading {0} documents...\n", numDocs.ToString());

    DateTime startTime = DateTime.Now;
    Console.WriteLine("Started at: {0} \n", startTime);
    Console.WriteLine("Creating {0} threads...\n", numThreads);

    // Creating a list to hold active tasks
    List<Task<IndexDocumentsResult>> uploadTasks = new
List<Task<IndexDocumentsResult>>();

    for (int i = 0; i < numDocs; i += batchSize)
    {
        List<Hotel> hotelBatch = hotels.GetRange(i, batchSize);
        var task = ExponentialBackoffAsync(searchClient, hotelBatch, i);
        uploadTasks.Add(task);
        Console.WriteLine("Sending a batch of {0} docs starting with doc
{1}...\n", batchSize, i);

        // Checking if we've hit the specified number of threads
        if (uploadTasks.Count >= numThreads)
        {
            Task<IndexDocumentsResult> firstTaskFinished = await
Task.WhenAny(uploadTasks);
            Console.WriteLine("Finished a thread, kicking off another...");
            uploadTasks.Remove(firstTaskFinished);
        }
    }

    // waiting for the remaining results to finish
    await Task.WhenAll(uploadTasks);

    DateTime endTime = DateTime.Now;

    TimeSpan runningTime = endTime - startTime;
    Console.WriteLine("\nEnded at: {0} \n", endTime);
    Console.WriteLine("Upload time total: {0}", runningTime);

    double timePerBatch = Math.Round(runningTime.TotalMilliseconds / (numDocs /
batchSize), 4);
    Console.WriteLine("Upload time per batch: {0} ms", timePerBatch);

    double timePerDoc = Math.Round(runningTime.TotalMilliseconds / numDocs, 4);
    Console.WriteLine("Upload time per document: {0} ms \n", timePerDoc);
}
```

This code uses **asynchronous calls** to a function `ExponentialBackoffAsync` that implements the backoff strategy. You call the function **using threads, for example, the number of cores your processor has**.

**When the maximum number of threads has been used, the code waits for any thread to finish. It then creates a new thread until all the documents are uploaded.**

---

## Exercise: Add to an index using the push API

You want to explore how to create an Azure AI Search index and upload documents to that index using C# code.

In this exercise, you'll clone an existing C# solution and run it to work out the optimal batch size to upload documents. You'll then use this batch size and upload documents effectively using a threaded approach.

**Note** To complete this exercise, you will need a Microsoft Azure subscription. If you don't already have one, you can sign up for a free trial at <https://azure.com/free>.

## Set up your Azure resources


To save you time, select this **Azure Resource Manager template to create resources** you'll need later in the exercise:

1. [Deploy resources to Azure](#) - select this link to create your Azure AI resources.

[Home](#) >

## Custom deployment ...


Deploy from a custom template

 New! Deployment Stacks let you manage the lifecycle of your deployments. Try it now →

**Basics**   Review + create

### Template



[Customized template](#)   
3 resources

 [Edit template](#)

 [Edit parameters](#)

 [Visualize](#)

### Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ

AI Subscription



Resource group \* ⓘ

(New) cog-search-language-exe

[Create new](#)

### Instance details

Region \* ⓘ

West Europe

Resource Prefix \* ⓘ

acs18245

Storage Account Type ⓘ

Standard\_LRS

Search Service Sku ⓘ

standard

Location ⓘ

West Europe

[Previous](#)

[Next](#)

[Review + create](#)

2. In **Resource group**, select **Create new**, name it **cog-search-language-exe**.
3. In **Region**, select a [supported region](#) that is close to you.
4. The **Resource Prefix** needs to be globally unique, enter a random numeric and lower-case character prefix, for example, **acs118245**.
5. In **Location**, select the same region you chose above.
6. Select **Review + create**.
7. Select **Create**.

- When deployment has finished, select **Go to resource group** to see all the resources that you've created.

Name	Type	Location
acs118245-cognitive-services	Azure AI services multi-service account	West Europe
acs118245-search-service	Search service	West Europe
acs118245str	Storage account	West Europe

## Copy Azure AI Search service REST API information

- In the list of resources, select the search service you created. In the above example **acs118245-search-service**.
- Copy the search service name into a text file.

Home > Resource groups > cog-search-language-exe > acs118245-search-service

- On the left, select **Keys**, then copy the **Primary admin key** into the same text file.

## Download example code

Open your the Azure Cloud Shell by selecting the Cloud Shell button at the top of the Azure portal.

**Note** If you're prompted to create an Azure Storage account select **Create storage**.

1. Once it has finished starting up, clone the following example code repository by running the following in your Cloud Shell:

```
git clone https://github.com/Azure-Samples/azure-search-dotnet-scale.git samples
```

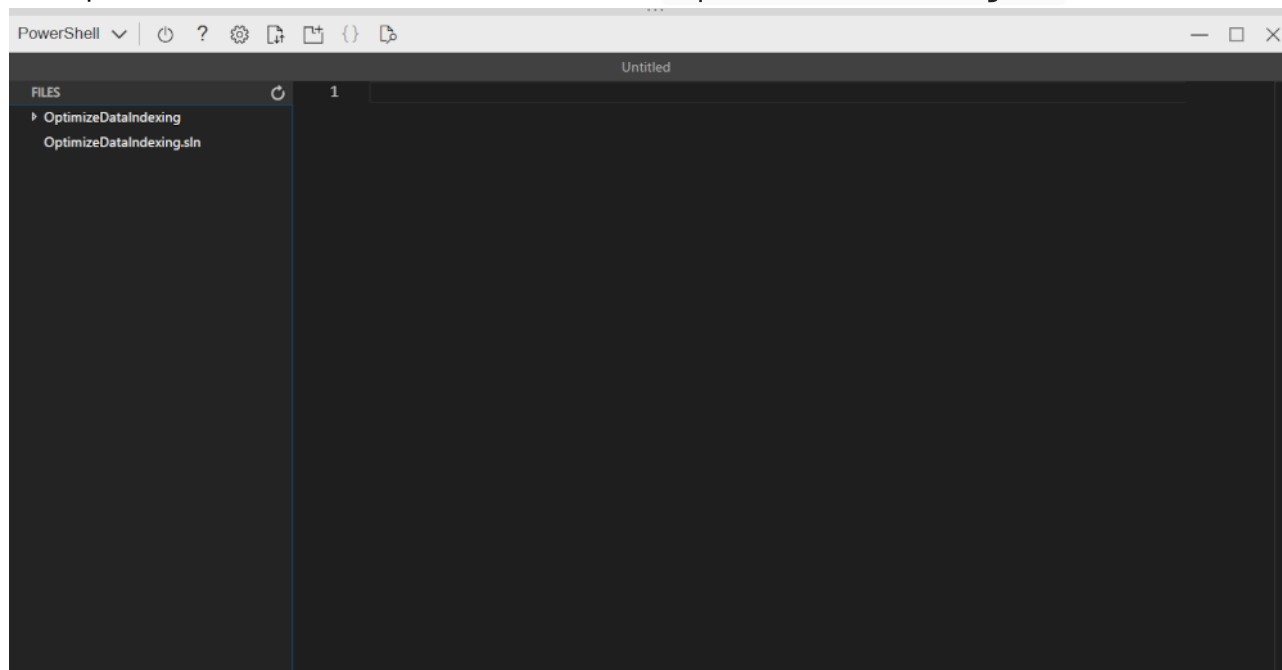
2. Change into the newly created directory by running:

```
cd samples
```

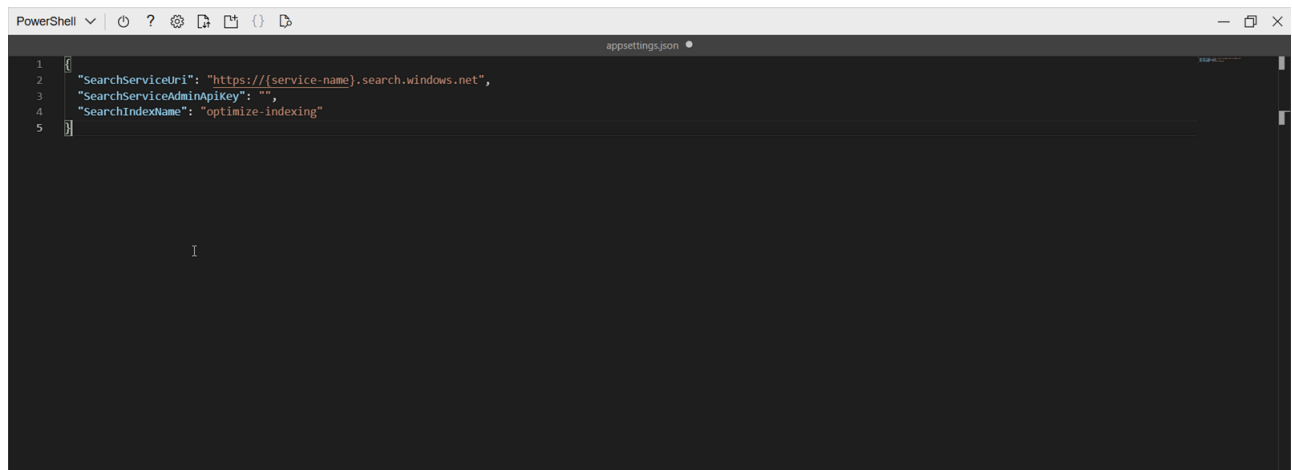
3. Then run:

```
code ./optimize-data-indexing/v11
```

4. This opens the code editor inside Cloud Shell at the `/optimize-data-indexing/v11` folder.



5. In the navigation on the left, expand the **OptimizeDataIndexing** folder, then select the **appsettings.json** file.



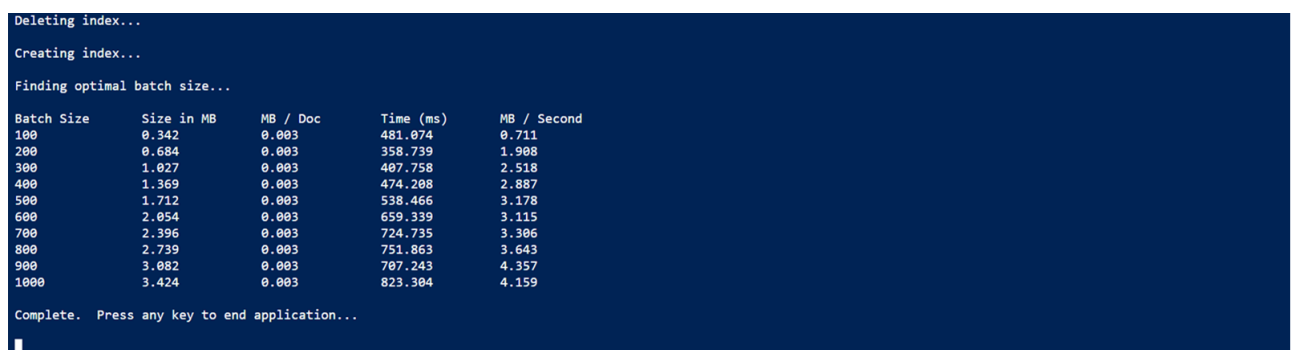
```
1 {
2   "SearchServiceUri": "https://{service-name}.search.windows.net",
3   "SearchServiceAdminApiKey": "",
4   "SearchIndexName": "optimize-indexing"
5 }
```

6. Paste in your search service name and **primary admin key**.

```
{
  "SearchServiceUri": "https://acs118245-search-service.search.windows.net",
  "SearchServiceAdminApiKey": "YOUR_SEARCH_SERVICE_KEY",
  "SearchIndexName": "optimize-indexing"
}
```

The settings file should look similar to the above.

7. Save your change by pressing **CTRL + S**.
8. Select the **OptimizeDataIndexing.csproj** file.
9. On the fifth line,  
change `<TargetFramework>netcoreapp3.1</TargetFramework>` to `<TargetFramework>net7.0</TargetFramework>`.
10. Save your change by pressing **CTRL + S**.
11. In the terminal, enter `cd ./optimize-data-indexing/v11/OptimizeDataIndexing` then press **Enter** to change into the correct directory.
12. Select the **Program.cs** file. Then, in the terminal, enter `dotnet run` and press **Enter**.



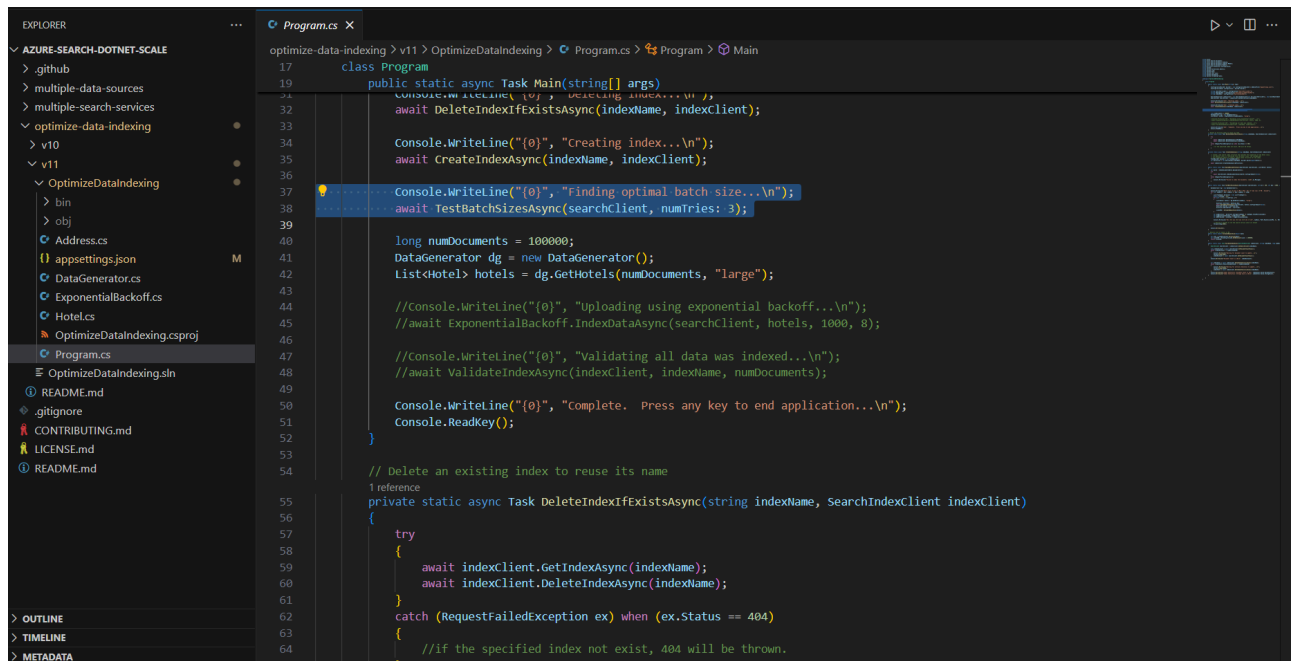
```
Deleting index...
Creating index...
Finding optimal batch size...
Batch Size      Size in MB      MB / Doc      Time (ms)      MB / Second
100             0.342           0.003         481.074         0.711
200             0.684           0.003         358.739         1.908
300             1.027           0.003         407.758         2.518
400             1.369           0.003         474.208         2.887
500             1.712           0.003         538.466         3.178
600             2.054           0.003         659.339         3.115
700             2.396           0.003         724.735         3.306
800             2.739           0.003         751.863         3.643
900             3.082           0.003         707.243         4.357
1000            3.424           0.003         823.304         4.159
Complete. Press any key to end application...
```

The output shows that in this case, the best performing batch size is 900 documents. As it reaches 3.688 MB per second.

## Edit the code to implement threading and a backoff and retry strategy

There's code commented out that's ready to change the app to use threads to upload documents to the search index.

1. Make sure you've selected **Program.cs**.



2. Comment out lines 38 and 39 like this:

```
//Console.WriteLine("{0}", "Finding optimal batch size...\n");
//await TestBatchSizesAsync(searchClient, numTries: 3);
```

3. Uncomment lines 41 to 49.

```
long numDocuments = 100000;
DataGenerator dg = new DataGenerator();
List<Hotel> hotels = dg.GetHotels(numDocuments, "large");

Console.WriteLine("{0}", "Uploading using exponential backoff...\n");
await ExponentialBackoff.IndexDataAsync(searchClient, hotels, 1000, 8);

Console.WriteLine("{0}", "Validating all data was indexed...\n");
await ValidateIndexAsync(indexClient, indexName, numDocuments);
```

The code that controls the batch size and number of threads is `await ExponentialBackoff.IndexDataAsync(searchClient, hotels, 1000, 8)`. The batch size is 1000 and the threads are eight.

```
17 0 references
18 class Program
19 {
20     0 references
21     public static async Task Main(string[] args)
22     {
23         IConfigurationBuilder builder = new ConfigurationBuilder().AddJsonFile("appsettings.json");
24         IConfigurationRoot configuration = builder.Build();
25
26         string searchServiceUri = configuration["SearchServiceUri"];
27         string adminApiKey = configuration["SearchServiceAdminApiKey"];
28         string indexName = configuration["SearchIndexName"];
29
30         SearchIndexClient indexClient = new SearchIndexClient(new Uri(searchServiceUri), new AzureKeyCredential(adminApiKey));
31         SearchClient searchClient = indexClient.GetSearchClient(indexName);
32
33         Console.WriteLine("{0}", "Deleting index...\n");
34         await DeleteIndexIfExistsAsync(indexName, indexClient);
35
36         Console.WriteLine("{0}", "Creating index...\n");
37         await CreateIndexAsync(indexName, indexClient);
38
39         //Console.WriteLine("{0}", "Finding optimal batch size...\n");
40         //await TestBatchSizesAsync(searchClient, numTries: 3);
41
42         long numDocuments = 100000;
43         DataGenerator dg = new DataGenerator();
44         List<Hotel> hotels = dg.GetHotels(numDocuments, "large");
45
46         Console.WriteLine("{0}", "Uploading using exponential backoff...\n");
47         await ExponentialBackoff.IndexDataAsync(searchClient, hotels, 1000, 8);
48
49         Console.WriteLine("{0}", "Validating all data was indexed...\n");
50         await ValidateIndexAsync(indexClient, indexName, numDocuments);
51
52         Console.WriteLine("{0}", "Complete. Press any key to end application...\n");
53         Console.ReadKey();
54     }
55 }
```

Your code should look like the above.

4. Save your changes, press **CTRL+S**.
5. Select your terminal, then press any key to end the running process if you haven't already.
6. Run `dotnet run` in the terminal.

```
Ended at: 9/1/2023 3:25:36 PM
Upload time total: 00:01:18.0220862
Upload time per batch: 780.2209 ms
Upload time per document: 0.7802 ms
Validating all data was indexed...
Waiting for document count to update...
Document Count is 100000
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Waiting for service statistics to update...
Index Statistics: Document Count is 100000
Index Statistics: Storage Size is 71453102
Complete. Press any key to end application...
```

The app will start eight threads, and then as each thread finishes writing a new message to the console:

```
Finished a thread, kicking off another...
Sending a batch of 1000 docs starting with doc 57000...
```



After 100,000 documents are uploaded, the app writes a summary (this might take a while to complete):

```
Ended at: 9/1/2023 3:25:36 PM

Upload time total: 00:01:18:0220862
Upload time per batch: 780.2209 ms
Upload time per document: 0.7802 ms

Validating all data was indexed...

Waiting for service statistics to update...

Document Count is 100000

Waiting for service statistics to update...

Index Statistics: Document Count is 100000
Index Statistics: Storage Size is 71453102
```


Explore the code in the `TestBatchSizesAsync` procedure to see how the code tests the batch size performance.

Explore the code in the `IndexDataAsync` procedure to see how the code manages threading.






Explore the code in the `ExponentialBackoffAsync` to see how the code implements an exponential backoff retry strategy.

You can search and verify that the documents have been added to the index in the Azure portal.







[Home](#) > [Resource groups](#) > [cog-search-language-exe](#) > [acs118245-search-service](#)

 **acs118245-search-service | Indexes** ☆ ...  
Search service

<< + Add index ▾ ↻ Refresh 🗑 Delete

 Overview  
 Activity log  
 Access control (IAM)  
 Tags  
 Diagnose and solve problems

Search management

 **Indexes**  
 Indexers  
 Data sources  
 Aliases  
 Skillsets  
 Debug sessions

Name	Document Count	Storage Size
<a href="#">optimize-indexing</a>	100,000	68.15 MB

## Delete exercise resources

Now that you've completed the exercise, delete all the resources you no longer need. Start with the code cloned to your machine. Then delete the Azure resources.

1. In the Azure portal, select **Resource groups**.
2. Select the resource group you've created for this exercise.
3. Select **Delete resource group**.
4. Confirm deletion then select **Delete**.
5. Select the resources you don't need, then select **Delete**.

---

## Knowledge Check

1. What is the limitation of using the Azure Search linked service as a sink in a copy data task? \*

- ☐ You can only upload one document at a time.
- ☒ The JSON can't contain complex data types like arrays.

✓ Correct. At the moment, the linked service only supports a limited number of field types.

- ☐ You have to define the index in the Azure portal first.

2. Which feature of the REST API would you use to upload documents into a search index? \*

- ☒ Index.

✓ Correct. You use the index REST API focused on documents.

- ☐ Indexer.

- ☐ Skillset.

3. Which response code will require you to implement a backoff strategy? \*

- ☐ 200 and 201.

- ☐ 404 and 501.

- ☒ 207 and 503.

✓ Correct. 503 is the response means the system is under heavy load and your request can't be processed at this time. 207 means that some documents succeeded, but at least one of them failed.

---

## Summary

In this module you've seen how to push data into an Azure AI Search index. Azure Data Factory has an Azure Search linked service that you can use as a sink in a copy data pipeline. The search service supports REST requests to push data into indexes.

You then used code to push data into an index using the best batch size and threading.

By completing this module, you learned how to:

- Use Azure Data Factory to copy data into an Azure AI Search Index.
- Use the Azure AI Search push API to add to an index from any external data source.

If you'd like to learn more about the search REST API, see [Azure AI Search Service REST](#).

If you'd like to learn more about indexing large datasets effectively, see [Index large data sets in Azure AI Search](#).

---

 Compiled by [Kenneth Leung](#) (2025)