Create a Custom Skill for Azure Cognitive Search

Azure Cognitive Search uses an enrichment pipeline of cognitive skills to extract AI-generated fields from documents and include them in a search index. There's a comprehensive set of built-in skills that you can use, but if you have a specific requirement that isn't met by these skills, you can create a custom skill.

In this exercise, you'll create a custom skill that tabulates the frequency of individual words in a document to generate a list of the top five most used words, and add it to a search solution for Margie's Travel - a fictitious travel agency.

Clone the repository for this course

If you have already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, open it in Visual Studio Code; otherwise, follow these steps to clone it now.

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/AI-102-AIEngineer` 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**.

Create Azure resources

> **Note**: If you have previously completed the **Create an Azure Cognitive Search solution** exercise, and still have these Azure resources in your subscription, you can skip this section and start at the **Create a search solution** section. Otherwise, follow the steps below to provision the required Azure resources.

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. View the **Resource groups** in your subscription.
3. If you are using a restricted subscription in which a resource group has been provided for you, select the resource group to view its properties. Otherwise, create a new resource group with a name of your choice, and go to it when it has been created.
4. On the **Overview** page for your resource group, note the **Subscription ID** and **Location**. You will need these values, along with the name of the resource group in subsequent steps.
5. In Visual Studio Code, expand the **23-custom-search-skill** folder and select **setup.cmd**. You will use this batch script to run the Azure command line interface (CLI) commands required to create the Azure resources you need.
6. Right-click the the **23-custom-search-skill** folder and select **Open in Integrated Terminal**.
7. In the terminal pane, enter the following command to establish an authenticated connection to your Azure subscription.

```
Code                                                    Copy

az login --output none
```

8. When prompted, sign into your Azure subscription. Then return to Visual Studio Code and wait for the sign-in process to complete.

9. Run the following command to list Azure locations.

```
az account list-locations -o table
```

10. In the output, find the **Name** value that corresponds with the location of your resource group (for example, for *East US* the corresponding name is *eastus*).
11. In the **setup.cmd** script, modify the **subscription_id**, **resource_group**, and **location** variable declarations with the appropriate values for your subscription ID, resource group name, and location name. Then save your changes.
12. In the terminal for the **23-custom-search-skill** folder, enter the following command to run the script:

```
setup
```

> **Note**: The Search CLI module is in preview, and may get stuck in the - *Running* .. process. If this happens for over 2 minutes, press CTRL+C to cancel the long-running operation, and then select **N** when asked if you want to terminate the script. It should then complete successfully. If the script fails, ensure you saved it with the correct variable names and try again.

13. When the script completes, review the output it displays and note the following information about your Azure resources (you will need these values later):
    - Storage account name
    - Storage connection string
    - Cognitive Services account
    - Cognitive Services key
    - Search service endpoint
    - Search service admin key
    - Search service query key
14. In the Azure portal, refresh the resource group and verify that it contains the Azure Storage account, Azure Cognitive Services resource, and Azure Cognitive Search resource.

Create a search solution

Now that you have the necessary Azure resources, you can create a search solution that consists of the following components:

- A **data source** that references the documents in your Azure storage container.
- A **skillset** that defines an enrichment pipeline of skills to extract AI-generated fields from the documents.
- An **index** that defines a searchable set of document records.
- An **indexer** that extracts the documents from the data source, applies the skillset, and populates the index.

In this exercise, you'll use the Azure Cognitive Search REST interface to create these components by submitting JSON requests.

1. In Visual Studio Code, in the **23-custom-search-skill** folder, expand the **create-search** folder and select **data_source.json**. This file contains a JSON definition for a data source named **margies-custom-data**.
2. Replace the **YOUR_CONNECTION_STRING** placeholder with the connection string for your Azure storage account, which should resemble the following:

```
DefaultEndpointsProtocol=https;AccountName=ai102str123;AccountKey=12345abcdefg...==;EndpointS
```

*You can find the connection string on the **Access keys** page for your storage account in the Azure portal.*

3. Save and close the updated JSON file.
4. In the **create-search** folder, open **skillset.json**. This file contains a JSON definition for a skillset named **margies-custom-skillset**.
5. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_COGNITIVE_SERVICES_KEY** placeholder with either of the keys for your cognitive services resources.

   *You can find the keys on the **Keys and Endpoint** page for your cognitive services resource in the Azure portal.*
6. Save and close the updated JSON file.
7. In the **create-search** folder, open **index.json**. This file contains a JSON definition for an index named **margies-custom-index**.
8. Review the JSON for the index, then close the file without making any changes.
9. In the **create-search** folder, open **indexer.json**. This file contains a JSON definition for an indexer named **margies-custom-indexer**.
10. Review the JSON for the indexer, then close the file without making any changes.
11. In the **create-search** folder, open **create-search.cmd**. This batch script uses the cURL utility to submit the JSON definitions to the REST interface for your Azure Cognitive Search resource.
12. Replace the **YOUR_SEARCH_URL** and **YOUR_ADMIN_KEY** variable placeholders with the **Url** and one of the **admin keys** for your Azure Cognitive Search resource.

    *You can find these values on the **Overview** and **Keys** pages for your Azure Cognitive Search resource in the Azure portal.*
13. Save the updated batch file.
14. Right-click the the **create-search** folder and select **Open in Integrated Terminal**.
15. In the terminal pane for the **create-search** folder, enter the following command run the batch script.

    Code Copy

    ```
    create-search
    ```
16. When the script completes, in the Azure portal, on the page for your Azure Cognitive Search resource, select the **Indexers** page and wait for the indexing process to complete.

    *You can select **Refresh** to track the progress of the indexing operation. It may take a minute or so to complete.*

Search the index

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

1. At the top of the blade for your Azure Cognitive Search resource, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter the following query string, and then select **Search**.

   Code Copy

   ```
   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)

Create an Azure Function for a custom skill

The search solution includes a number of built-in cognitive skills that enrich the index with information from the documents, such as the sentiment scores and lists of key phrases seen in the previous task.

You can enhance the index further by creating custom skills. For example, it might be useful to identify the words that are used most frequently in each document, but no built-in skill offers this functionality.

To implement the word count functionality as a custom skill, you'll create an Azure Function in your preferred language.

1. In Visual Studio Code, view the Azure Extensions tab (**⊞**), and verify that the **Azure Functions** extension is installed. This extension enables you to create and deploy Azure Functions from Visual Studio Code.

2. On Azure tab (**Δ**), in the **Azure Functions** pane, create a new project (📂) with the following settings, depending on your preferred language:

   ### **C#**

   - **Folder**: Browse to **23-custom-search-skill/C-Sharp/wordcount**
   - **Language**: C#
   - **Template**: HTTP trigger
   - **Function name**: wordcount
   - **Namespace**: margies.search
   - **Authorization level**: Function

   ### **Python**

   - **Folder**: Browse to **23-custom-search-skill/Python/wordcount**
   - **Language**: Python
   - **Virtual environment**: Skip virtual environment
   - **Template**: HTTP trigger
   - **Function name**: wordcount
   - **Authorization level**: Function

   *If you are prompted to overwrite **launch.json**, do so!*

3. Switch back to the **Explorer** (🗐) tab and verify that the **wordcount** folder now contains the code files for your Azure Function.

   *If you chose Python, the code files may be in a subfolder, also named **wordcount***
4. The main code file for your function should have been opened automatically. If not, open the appropriate file for your chosen language:

   - **C#**: wordcount.cs
   - **Python**: __init__.py
5. Replace the entire contents of the file with the following code for your chosen language:

C#

```
Code
```

Copy

```
using System.IO;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Text.RegularExpressions;
using System.Linq;

namespace margies.search
{
    public static class wordcount
    {

        //define classes for responses
        private class WebApiResponseError
        {
            public string message { get; set; }
        }

        private class WebApiResponseWarning
        {
            public string message { get; set; }
        }

        private class WebApiResponseRecord
        {
            public string recordId { get; set; }
            public Dictionary<string, object> data { get; set; }
            public List<WebApiResponseError> errors { get; set; }
            public List<WebApiResponseWarning> warnings { get; set; }
        }

        private class WebApiEnricherResponse
        {
            public List<WebApiResponseRecord> values { get; set; }
        }

        //function for custom skill
        [FunctionName("wordcount")]
        public static IActionResult Run(
            [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)]HttpRequest req,
ILogger log)
        {
            log.LogInformation("Function initiated.");

            string recordId = null;
            string originalText = null;

            string requestBody = new StreamReader(req.Body).ReadToEnd();
            dynamic data = JsonConvert.DeserializeObject(requestBody);

            // Validation
            if (data?.values == null)
            {
                return new BadRequestObjectResult(" Could not find values array");
```

```
            }
            if (data?.values.HasValues == false || data?.values.First.HasValues == false)
            {
                return new BadRequestObjectResult("Could not find valid records in values
array");
            }

            WebApiEnricherResponse response = new WebApiEnricherResponse();
            response.values = new List<WebApiResponseRecord>();
            foreach (var record in data?.values)
            {
                recordId = record.recordId?.Value as string;
                originalText = record.data?.text?.Value as string;

                if (recordId == null)
                {
                    return new BadRequestObjectResult("recordId cannot be null");
                }

                // Put together response.
                WebApiResponseRecord responseRecord = new WebApiResponseRecord();
                responseRecord.data = new Dictionary<string, object>();
                responseRecord.recordId = recordId;
                responseRecord.data.Add("text", Count(originalText));

                response.values.Add(responseRecord);
            }

            return (ActionResult)new OkObjectResult(response);
        }


            public static string RemoveHtmlTags(string html)
        {
            string htmlRemoved = Regex.Replace(html, @"<script[^>]*>[\s\S]*?</script>|<[^>]+>| ",
" ").Trim();
            string normalised = Regex.Replace(htmlRemoved, @"\s{2,}", " ");
            return normalised;
        }

        public static List<string> Count(string text)
        {

            //remove html elements
            text=text.ToLowerInvariant();
            string html = RemoveHtmlTags(text);

            //split into list of words
            List<string> list = html.Split(" ").ToList();

            //remove any non alphabet characters
            var onlyAlphabetRegEx = new Regex(@"^[A-z]+$");
            list = list.Where(f => onlyAlphabetRegEx.IsMatch(f)).ToList();

            //remove stop words
            string[] stopwords = { "", "i", "me", "my", "myself", "we", "our", "ours",
"ourselves", "you",
                    "you're", "you've", "you'll", "you'd", "your", "yours", "yourself",
                    "yourselves", "he", "him", "his", "himself", "she", "she's", "her",
```

C#

```
                    "hers", "herself", "it", "it's", "its", "itself", "they", "them",
                    "their", "theirs", "themselves", "what", "which", "who", "whom",
                    "this", "that", "that'll", "these", "those", "am", "is", "are", "was",
                    "were", "be", "been", "being", "have", "has", "had", "having", "do",
                    "does", "did", "doing", "a", "an", "the", "and", "but", "if", "or",
                    "because", "as", "until", "while", "of", "at", "by", "for", "with",
                    "about", "against", "between", "into", "through", "during", "before",
                    "after", "above", "below", "to", "from", "up", "down", "in", "out",
                    "on", "off", "over", "under", "again", "further", "then", "once", "here",
                    "there", "when", "where", "why", "how", "all", "any", "both", "each",
                    "few", "more", "most", "other", "some", "such", "no", "nor", "not",
                    "only", "own", "same", "so", "than", "too", "very", "s", "t", "can",
                    "will", "just", "don", "don't", "should", "should've", "now", "d", "ll",
                    "m", "o", "re", "ve", "y", "ain", "aren", "aren't", "couldn", "couldn't",
                    "didn", "didn't", "doesn", "doesn't", "hadn", "hadn't", "hasn", "hasn't",
                    "haven", "haven't", "isn", "isn't", "ma", "mightn", "mightn't", "mustn",
                    "mustn't", "needn", "needn't", "shan", "shan't", "shouldn", "shouldn't",
"wasn",
                    "wasn't", "weren", "weren't", "won", "won't", "wouldn", "wouldn't"};
            list = list.Where(x => x.Length > 2).Where(x => !stopwords.Contains(x)).ToList();

            //get distict words by key and count, and then order by count.
            var keywords = list.GroupBy(x => x).OrderByDescending(x => x.Count());
            var klist = keywords.ToList();

            // return the top 10 words
            var numofWords = 10;
            if(klist.Count<10)
                numofWords=klist.Count;
            List<string> resList = new List<string>();
            for (int i = 0; i < numofWords; i++)
            {
                resList.Add(klist[i].Key);
            }
            return resList;
        }
    }
}
```

Python

Code

Copy

```
import logging
import os
import sys
import json
from string import punctuation
from collections import Counter
import azure.functions as func


def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Wordcount function initiated.')

    # The result will be a "values" bag
    result = {
        "values": []
    }
    statuscode = 200

    # We're going to exclude words from this list in the word counts
    stopwords = ['', 'i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you',
                 "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself',
                 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her',
                 'hers', 'herself', 'it', "it's", 'its', 'itself', 'they', 'them',
                 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom',
                 'this', 'that', "that'll", 'these', 'those', 'am', 'is', 'are', 'was',
                 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do',
                 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or',
                 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with',
                 'about', 'against', 'between', 'into', 'through', 'during', 'before',
                 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out',
                 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'once', 'here',
                 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each',
                 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not',
                 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can',
                 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll',
                 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'couldn', "couldn't",
                 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't",
                 'haven', "haven't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn',
                 "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "shouldn't", 'wasn',
                 "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]

    try:
        values = req.get_json().get('values')
        logging.info(values)

        for rec in values:
            # Construct the basic JSON response for this record
            val = {
                    "recordId": rec['recordId'],
                    "data": {
                        "text":None
                    },
                    "errors": None,
                    "warnings": None
                }
            try:
                # get the text to be processed from the input record
                txt = rec['data']['text']
```

```
                # remove numeric digits
                txt = ''.join(c for c in txt if not c.isdigit())
                # remove punctuation and make lower case
                txt = ''.join(c for c in txt if c not in punctuation).lower()
                # remove stopwords
                txt = ' '.join(w for w in txt.split() if w not in stopwords)
                # Count the words and get the most common 10
                wordcount = Counter(txt.split()).most_common(10)
                words = [w[0] for w in wordcount]
                # Add the top 10 words to the output for this text record
                val["data"]["text"] = words
            except:
                # An error occured for this text record, so add lists of errors and warning
                val["errors"] =[{"message": "An error occurred processing the text."}]
                val["warnings"] = [{"message": "One or more inputs failed to process."}]
            finally:
                # Add the value for this record to the response
                result["values"].append(val)
    except Exception as ex:
        statuscode = 500
        # A global error occurred, so return an error response
        val = {
                "recordId": None,
                "data": {
                    "text":None
                },
                "errors": [{"message": ex.args}],
                "warnings": [{"message": "The request failed to process."}]
            }
        result["values"].append(val)
    finally:
        # Return the response
        return func.HttpResponse(body=json.dumps(result), mimetype="application/json",
status_code=statuscode)
```

1. Save the updated file.

2. Right-click the **wordcount** folder containing your code files and select **Deploy to Function App**. Then deploy the function with the following language-specific settings (signing into Azure if prompted):

   ### **C#**

   - **Subscription** (if prompted): Select your Azure subscription.
   - **Function**: Create a new Function App in Azure (Advanced)
   - **Function App Name**: Enter a globally unique name.
   - **Runtime**: .NET Core 3.1
   - **OS**: Linux
   - **Hosting plan**: Consumption
   - **Resource group**: The resource group containing your Azure Cognitive Search resource.
     - Note: If this resource group already contains a Windows-based web app, you will not be able to deploy a Linux-based function there. Either delete the existing web app or deploy the function to a different resource group.
   - **Storage account**: The storage count where the Margie's Travel docs are stored.
   - **Application Insights**: Skip for now

   *Visual Studio Code will deploy the compiled version of the function (in the **bin** subfolder) based on the configuration settings in the **.vscode** folder that were saved when you created the function project.*

   ### **Python**

- **Subscription** (if prompted): Select your Azure subscription.
- **Function**: Create a new Function App in Azure (Advanced)
- **Function App Name**: Enter a globally unique name.
- **Runtime**: Python 3.8
- **Hosting plan**: Consumption
- **Resource group**: The resource group containing your Azure Cognitive Search resource.
    - Note: If this resource group already contains a Windows-based web app, you will not be able to deploy a Linux-based function there. Either delete the existing web app or deploy the function to a different resource group.
- **Storage account**: The storage count where the Margie's Travel docs are stored.
- **Application Insights**: Skip for now

3. Wait for Visual Studio Code to deploy the function. A notification will appear when deployment is complete.

Test the function

Now that you've deployed the function to Azure, you can test it in the Azure portal.

1. Open the Azure portal, and browse to the resource group where you created the function app. Then open the app service for your function app.
2. In the blade for your app service, on the **Functions** page, open the **wordcount** function.
3. On the **wordcount** function blade, view the **Code + Test** page and open the **Test/Run** pane.
4. In the **Test/Run** pane, replace the existing **Body** with the following JSON, which reflects the schema expected by an Azure Cognitive Search skill in which records containing data for one or more documents are submitted for processing:

Code Copy

```
{
    "values": [
        {
            "recordId": "a1",
            "data":
            {
            "text":  "Tiger, tiger burning bright in the darkness of the night.",
            "language": "en"
            }
        },
        {
            "recordId": "a2",
            "data":
            {
            "text":  "The rain in spain stays mainly in the plains! That's where you'll
find the rain!",
            "language": "en"
            }
        }
    ]
}
```

5. Click **Run** and view the HTTP response content that is returned by your function. This reflects the schema expected by Azure Cognitive Search when consuming a skill, in which a response for each document is returned. In this case, the response consists of up to 10 terms in each document in descending order of how frequently they appear:

Code Copy

```
{
"values": [
    {
    "recordId": "a1",
    "data": {
        "text": [
        "tiger",
        "burning",
        "bright",
        "darkness",
        "night"
        ]
    },
    "errors": null,
    "warnings": null
    },
    {
    "recordId": "a2",
    "data": {
        "text": [
        "rain",
        "spain",
        "stays",
        "mainly",
        "plains",
        "thats",
        "youll",
        "find"
        ]
    },
    "errors": null,
    "warnings": null
    }
]
}
```


6. Close the **Test/Run** pane and in the **wordcount** function blade, click **Get function URL**. Then copy the URL for the default key to the clipboard. You'll need this in the next procedure.

Add the custom skill to the search solution

Now you need to include your function as a custom skill in the search solution skillset, and map the results it produces to a field in the index.

1. In Visual Studio Code, in the **23-custom-search-skill/update-search** folder, open the **update-skillset.json** file. This contains the JSON definition of a skillset.
2. Review the skillset definition. It includes the same skills as before, as well as a new **WebApiSkill** skill named **get-top-words**.
3. Edit the **get-top-words** skill definition to set the **uri** value to the URL for your Azure function (which you copied to the clipboard in the previous procedure), replacing **YOUR-FUNCTION-APP-URL**.
4. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_COGNITIVE_SERVICES_KEY** placeholder with either of the keys for your cognitive services resources.

   *You can find the keys on the **Keys and Endpoint** page for your cognitive services resource in the Azure portal.*
5. Save and close the updated JSON file.

6. In the **update-search** folder, open **update-index.json**. This file contains the JSON definition for the **margies-custom-index** index, with an additional field named **top_words** at the bottom of the index definition.
7. Review the JSON for the index, then close the file without making any changes.
8. In the **update-search** folder, open **update-indexer.json**. This file contains a JSON definition for the **margies-custom-indexer**, with an additional mapping for the **top_words** field.
9. Review the JSON for the indexer, then close the file without making any changes.
10. In the **update-search** folder, open **update-search.cmd**. This batch script uses the cURL utility to submit the updated JSON definitions to the REST interface for your Azure Cognitive Search resource.
11. Replace the **YOUR_SEARCH_URL** and **YOUR_ADMIN_KEY** variable placeholders with the **Url** and one of the **admin keys** for your Azure Cognitive Search resource.

    *You can find these values on the **Overview** and **Keys** pages for your Azure Cognitive Search resource in the Azure portal.*
12. Save the updated batch file.
13. Right-click the the **update-search** folder and select **Open in Integrated Terminal**.
14. In the terminal pane for the **update-search** folder, enter the following command run the batch script.

    Code Copy

    ```
    update-search
    ```
15. When the script completes, in the Azure portal, on the page for your Azure Cognitive Search resource, select the **Indexers** page and wait for the indexing process to complete.

    *You can select **Refresh** to track the progress of the indexing operation. It may take a minute or so to complete.*

Search the index

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

1. At the top of the blade for your Azure Cognitive Search resource, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter the following query string, and then select **Search**.

    Code Copy

    ```
    search=Las Vegas&$select=url,top_words
    ```

    This query retrieves the **url** and **top_words** fields for all documents that mention *Las Vegas*.

More information

To learn more about creating custom skills for Azure Cognitive Search, see the Azure Cognitive Search documentation.