Provision an Azure AI Language resource

Clone the repository for this course

Configure your application

Add code to connect to your Azure AI Language resource

Add code to detect language

Add code to evaluate sentiment

Add code to identify key phrases

Add code to extract entities

Add code to extract linked entities

Clean up resources

More information

Analyze Text

Azure AI Language supports analysis of text, including language detection, sentiment analysis, key phrase extraction, and entity recognition.

For example, suppose a travel agency wants to process hotel reviews that have been submitted to the company's web site. By using the Azure AI Language, they can determine the language each review is written in, the sentiment (positive, neutral, or negative) of the reviews, key phrases that might indicate the main topics discussed in the review, and named entities, such as places, landmarks, or people mentioned in the reviews. In this exercise, you'll use the Azure AI Language Python SDK for text analytics to implement a simple hotel review application based on this example.

While this exercise is based on Python, you can develop text analytics applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **30** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource in your Azure subscription.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select **Continue to create your resource**.
5. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Responsible AI Notice**: Agree.
6. Select **Review + create**, then select **Create** to provision the resource.
7. Wait for deployment to complete, and then go to the deployed resource.
8. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Clone the repository for this course

You'll develop your code using Cloud Shell from the Azure Portal. The code files for your app have been provided in a GitHub repo.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

> **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/01-analyze-text/Python/text-analysis
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **text-analysis** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**text-analysis.py**). The text your application will analyze is in the **reviews** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-textanalytics==5.3.0
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal)
5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to connect to your Azure AI Language resource

1. Enter the following command to edit the application code file:

```
code text-analysis.py
```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a client for the Text Analysis API:

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

```
python text-analysis.py
```

6. Observe the output as the code should run without error, displaying the contents of each review text file in the **reviews** folder. The application successfully creates a client for the Text Analytics API but doesn't make use of it. We'll fix that in the next section.

Add code to detect language

Now that you have created a client for the API, let's use it to detect the language in which each review is written.

1. In the code editor, find the comment **Get language**. Then add the code necessary to detect the language in each review document:

```
# Get language
detectedLanguage = ai_client.detect_language(documents=[text])[0]
print('\nLanguage: {}'.format(detectedLanguage.primary_language.name))
```

> **Note**: *In this example, each review is analyzed individually, resulting in a separate call to the service for each file. An alternative approach is to create a collection of documents and pass them to the service in a single call. In both approaches, the response from the service consists of a collection of documents; which is why in the Python code above, the index of the first (and only) document in the response ([0]) is specified.*

2. Save your changes. Then re-run the program.
3. Observe the output, noting that this time the language for each review is identified.

Add code to evaluate sentiment

Sentiment analysis is a commonly used technique to classify text as *positive* or *negative* (or possible *neutral* or *mixed*). It's commonly used to analyze social media posts, product reviews, and other items where the sentiment of the text may provide useful insights.

1. In the code editor, find the comment **Get sentiment**. Then add the code necessary to detect the sentiment of each review document:

Code Copy

```
# Get sentiment
sentimentAnalysis = ai_client.analyze_sentiment(documents=[text])[0]
print("\nSentiment: {}".format(sentimentAnalysis.sentiment))
```

2. Save your changes. Then close the code editor and re-run the program.
3. Observe the output, noting that the sentiment of the reviews is detected.

Add code to identify key phrases

It can be useful to identify key phrases in a body of text to help determine the main topics that it discusses.

1. In the code editor, find the comment **Get key phrases**. Then add the code necessary to detect the key phrases in each review document:

Code Copy

```
# Get key phrases
phrases = ai_client.extract_key_phrases(documents=[text])[0].key_phrases
if len(phrases) > 0:
     print("\nKey Phrases:")
     for phrase in phrases:
          print('\t{}'.format(phrase))
```

2. Save your changes and re-run the program.
3. Observe the output, noting that each document contains key phrases that give some insights into what the review is about.

Add code to extract entities

Often, documents or other bodies of text mention people, places, time periods, or other entities. The text Analytics API can detect multiple categories (and subcategories) of entity in your text.

1. In the code editor, find the comment **Get entities**. Then, add the code necessary to identify entities that are mentioned in each review:

Code Copy

```
# Get entities
entities = ai_client.recognize_entities(documents=[text])[0].entities
if len(entities) > 0:
     print("\nEntities")
     for entity in entities:
         print('\t{} ({})'.format(entity.text, entity.category))
```

2. Save your changes and re-run the program.
3. Observe the output, noting the entities that have been detected in the text.

Add code to extract linked entities

In addition to categorized entities, the Text Analytics API can detect entities for which there are known links to data sources, such as Wikipedia.

1. In the code editor, find the comment **Get linked entities**. Then, add the code necessary to identify linked entities that are mentioned in each review:

Code Copy

```
# Get linked entities
entities = ai_client.recognize_linked_entities(documents=[text])[0].entities
if len(entities) > 0:
     print("\nLinks")
     for linked_entity in entities:
         print('\t{} ({})'.format(linked_entity.name, linked_entity.url))
```

2. Save your changes and re-run the program.
3. Observe the output, noting the linked entities that are identified.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

For more information about using **Azure AI Language**, see the documentation.

Create a Question Answering Solution

Provision an Azure AI Language resource

Create a question answering project

Add sources to the knowledge base

Edit the knowledge base

Train and test the knowledge base

Deploy the knowledge base

Prepare to develop an app in Cloud Shell

Configure your application

Add code to user your knowledge base

Clean up resources

More information

One of the most common conversational scenarios is providing support through a knowledge base of frequently asked questions (FAQs). Many organizations publish FAQs as documents or web pages, which works well for a small set of question and answer pairs, but large documents can be difficult and time-consuming to search.

Azure AI Language includes a *question answering* capability that enables you to create a knowledge base of question and answer pairs that can be queried using natural language input, and is most commonly used as a resource that a bot can use to look up answers to questions submitted by users. In this exercise, you'll use the Azure AI Language Python SDK for text analytics to implement a simple question answering application.

While this exercise is based on Python, you can develop question answering applications using multiple language-specific SDKs; including:

- Azure AI Language Service Question Answering client library for Python
- Azure AI Language Service Question Answering client library for .NET

This exercise takes approximately **20** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, to create and host a knowledge base for question answering, you need to enable the **Question Answering** feature.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select the **Custom question answering** block. Then select **Continue to create your resource**. You will need to enter the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Choose or create a resource group*.
    - **Region**: *Choose any available location*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Azure Search region**: *Choose a location in the same global region as your Language resource*
    - **Azure Search pricing tier**: Free (F) (*If this tier is not available, select Basic (B)*)
    - **Responsible AI Notice**: *Agree*
5. Select **Create + review**, then select **Create**.

> **NOTE** Custom Question Answering uses Azure Search to index and query the knowledge base of questions and answers.

6. Wait for deployment to complete, and then go to the deployed resource.
7. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Create a question answering project

To create a knowledge base for question answering in your Azure AI Language resource, you can use the Language Studio portal to create a question answering project. In this case, you'll create a knowledge base containing questions and answers about Microsoft Learn.

1. In a new browser tab, go to the Language Studio portal at https://language.cognitive.azure.com/ and sign in using the Microsoft account associated with your Azure subscription.
2. If you're prompted to choose a Language resource, select the following settings:
    - **Azure Directory**: The Azure directory containing your subscription.
    - **Azure subscription**: Your Azure subscription.
    - **Resource type**: Language
    - **Resource name**: The Azure AI Language resource you created previously.

    If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

    a. On the bar at the top if the page, select the **Settings (⚙)** button.
    b. On the **Settings** page, view the **Resources** tab.
    c. Select the language resource you just created, and click **Switch resource**.
    d. At the top of the page, click **Language Studio** to return to the Language Studio home page.
3. At the top of the portal, in the **Create new** menu, select **Custom question answering**.
4. In the *__Create a project__ wizard, on the **Choose language setting** page, select the option to **Select the language for all projects**, and select **English** as the language. Then select **Next**.
5. On the **Enter basic information** page, enter the following details:
    - **Name** `LearnFAQ`
    - **Description**: `FAQ for Microsoft Learn`
    - **Default answer when no answer is returned**: `Sorry, I don't understand the question`
6. Select **Next**.
7. On the **Review and finish** page, select **Create project**.

Add sources to the knowledge base

You can create a knowledge base from scratch, but it's common to start by importing questions and answers from an existing FAQ page or document. In this case, you'll import data from an existing FAQ web page for Microsoft Learn, and you'll also import some pre-defined "chit chat" questions and answers to support common conversational exchanges.

1. On the **Manage sources** page for your question answering project, in the **+ Add source** list, select **URLs**. Then in the **Add URLs** dialog box, select **+ Add url** and set the following name and URL before you select **Add all** to add it to the knowledge base:
    - **Name**: `Learn FAQ Page`
    - **URL**: `https://learn.microsoft.com/en-us/training/support/faq?pivots=general`
2. On the **Manage sources** page for your question answering project, in the **+ Add source** list, select **Chitchat**. The in the **Add chit chat** dialog box, select **Friendly** and select **Add chit chat**.

Edit the knowledge base

Your knowledge base has been populated with question and answer pairs from the Microsoft Learn FAQ, supplemented with a set of conversational *chit-chat* question and answer pairs. You can extend the knowledge base by adding additional question and answer pairs.

1. In your **LearnFAQ** project in Language Studio, select the **Edit knowledge base** page to see the existing question and answer pairs (if some tips are displayed, read them and choose **Got it** to dismiss them, or select **Skip all**)

2. In the knowledge base, on the **Question answer pairs** tab, select **+**, and create a new question answer pair with the following settings:

   - **Source**: `https://learn.microsoft.com/en-us/training/support/faq?pivots=general`
   - **Question**: `What are the different types of modules on Microsoft Learn?`
   - **Answer**:

     ```
     Microsoft Learn offers various types of training modules, including role-based learning paths, product-specific modules, and hands-on labs. Each module contains units with lessons and knowledge checks to help you learn at your own pace.
     ```

3. Select **Done**.

4. In the page for the **What are the different types of modules on Microsoft Learn?** question that is created, expand **Alternate questions**. Then add the alternate question `How are training modules organized?`.

   In some cases, it makes sense to enable the user to follow up on an answer by creating a *multi-turn* conversation that enables the user to iteratively refine the question to get to the answer they need.

5. Under the answer you entered for the module types question, expand **Follow-up prompts** and add the following follow-up prompt:

   - **Text displayed in the prompt to the user**: `Learn more about training`.
   - Select the **Create link to new pair** tab, and enter this text:

     ```
     You can explore modules and learning paths on the [Microsoft Learn training page](https://learn.microsoft.com/training/).
     ```

   - Select **Show in contextual flow only**. This option ensures that the answer is only ever returned in the context of a follow-up question from the original module types question.

6. Select **Add prompt**.

Train and test the knowledge base

Now that you have a knowledge base, you can test it in Language Studio.

1. Save the changes to your knowledge base by selecting the **Save** button under the **Question answer pairs** tab on the left.
2. After the changes have been saved, select the **Test** button to open the test pane.
3. In the test pane, at the top, deselect **Include short answer response** (if not already unselected). Then at the bottom enter the message `Hello`. A suitable response should be returned.
4. In the test pane, at the bottom enter the message `What is Microsoft Learn?`. An appropriate response from the FAQ should be returned.
5. Enter the message `Thanks!` An appropriate chit-chat response should be returned.
6. Enter the message `What are the different types of modules on Microsoft Learn?`. The answer you created should be returned along with a follow-up prompt link.
7. Select the **Learn more about training** follow-up link. The follow-up answer with a link to the training page should be returned.
8. When you're done testing the knowledge base, close the test pane.

Deploy the knowledge base

The knowledge base provides a back-end service that client applications can use to answer questions. Now you are ready to publish your knowledge base and access its REST interface from a client.

1. In the **LearnFAQ** project in Language Studio, select the **Deploy knowledge base** page from the navigation menu on the left.

2. At the top of the page, select **Deploy**. Then select **Deploy** to confirm you want to deploy the knowledge base.
3. When deployment is complete, select **Get prediction URL** to view the REST endpoint for your knowledge base and note that the sample request includes parameters for:
    - **projectName**: The name of your project (which should be *LearnFAQ*)
    - **deploymentName**: The name of your deployment (which should be *production*)
4. Close the prediction URL dialog box.

Prepare to develop an app in Cloud Shell

You'll develop your question answering app using Cloud Shell in the Azure portal. The code files for your app have been provided in a GitHub repo.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

    > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

    Code Copy

    ```
    rm -r mslearn-ai-language -f
    git clone https://github.com/microsoftlearning/mslearn-ai-language
    ```

    > **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

    Code Copy

    ```
    cd mslearn-ai-language/Labfiles/02-qna/Python/qna-app
    ```

Configure your application

1. In the command line pane, run the following command to view the code files in the **qna-app** folder:

    Code Copy

    ```
    ls -a -l
    ```

    The files include a configuration file (**.env**) and a code file (**qna-app.py**).

2. Create a Python virtual environment and install the Azure AI Language Question Answering SDK package and other required packages by running the following command:

    Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-language-questionanswering
```

3. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

4. In the code file, update the configuration values it contains to reflect the **endpoint** and an authentication **key** for the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal). The project name and deployment name for your deployed knowledge base should also be in this file.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to user your knowledge base

1. Enter the following command to edit the application code file:

Code Copy

```
code qna-app.py
```

2. Review the existing code. You will add code to work with your knowledge base.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. In the code file, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Question Answering SDK:

Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.language.questionanswering import QuestionAnsweringClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a question answering client:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = QuestionAnsweringClient(endpoint=ai_endpoint, credential=credential)
```

5. In the code file, find the comment **Submit a question and display the answer**, and add the following code to repeatedly read questions from the command line, submit them to the service, and display details of the answers:

Code Copy

```
# Submit a question and display the answer
user_question = ''
while True:
    user_question = input('\nQuestion:\n')
    if user_question.lower() == "quit":
        break
    response = ai_client.get_answers(question=user_question,
                                     project_name=ai_project_name,
                                     deployment_name=ai_deployment_name)
    for candidate in response.answers:
        print(candidate.answer)
        print("Confidence: {}".format(candidate.confidence))
        print("Source: {}".format(candidate.source))
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python qna-app.py
```

7. When prompted, enter a question to be submitted to your question answering project; for example `What is a learning path?`.
8. Review the answer that is returned.
9. Ask more questions. When you're done, enter `quit`.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

To learn more about question answering in Azure AI Language, see the Azure AI Language documentation.

Provision an Azure AI Language resource

Create a conversational language understanding project

Add entities

Use the model from a client app

Clean up resources

More information

Create a language understanding model with the Language service

The Azure AI Language service enables you to define a *conversational language understanding* model that applications can use to interpret natural language *utterances* from users (text or spoken input), predict the users *intent* (what they want to achieve), and identify any *entities* to which the intent should be applied.

For example, a conversational language model for a clock application might be expected to process input such as:

What is the time in London?

This kind of input is an example of an *utterance* (something a user might say or type), for which the desired *intent* is to get the time in a specific location (an *entity*); in this case, London.

> **NOTE** The task of a conversational language model is to predict the user's intent and identify any entities to which the intent applies. It is not the job of a conversational language model to actually perform the actions required to satisfy the intent. For example, a clock application can use a conversational language model to discern that the user wants to know the time in London; but the client application itself must then implement the logic to determine the correct time and present it to the user.

In this exercise, you'll use the Azure AI Language service to create a conversational language understand model, and use the Python SDK to implement a client app that uses it.

While this exercise is based on Python, you can develop conversational understanding applications using multiple language-specific SDKs; including:

- Azure AI Conversations client library for Python
- Azure AI Conversations client library for .NET
- Azure AI Conversations client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource in your Azure subscription.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - China East 2
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North

    - UK South
    - West Europe
    - West US 2
    - West US 3
  - **Name**: *Enter a unique name*.
  - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
  - **Responsible AI Notice**: Agree.
5. Select **Review + create**, then select **Create** to provision the resource.
6. Wait for deployment to complete, and then go to the deployed resource.
7. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Create a conversational language understanding project

Now that you have created an authoring resource, you can use it to create a conversational language understanding project.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.
2. If prompted to choose a Language resource, select the following settings:

   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.
   - **Language resource**: The Azure AI Language resource you created previously.

   If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

   a. On the bar at the top of the page, select the **Settings (⚙)** button.
   b. On the **Settings** page, view the **Resources** tab.
   c. Select the language resource you just created, and click **Switch resource**.
   d. At the top of the page, click **Language Studio** to return to the Language Studio home page

3. At the top of the portal, in the **Create new** menu, select **Conversational language understanding**.
4. In the **Create a project** dialog box, on the **Enter basic information** page, enter the following details and then select **Next**:

   - **Name**: `Clock`
   - **Utterances primary language**: English
   - **Enable multiple languages in project?**: *Unselected*
   - **Description**: `Natural language clock`

5. On the **Review and finish** page, select **Create**.

Create intents

The first thing we'll do in the new project is to define some intents. The model will ultimately predict which of these intents a user is requesting when submitting a natural language utterance.

> **Tip**: When working on your project, if some tips are displayed, read them and select **Got it** to dismiss them, or select **Skip all**.

1. On the **Schema definition** page, on the **Intents** tab, select **+ Add** to add a new intent named `GetTime`.
2. Verify that the **GetTime** intent is listed (along with the default **None** intent). Then add the following additional intents:

   - `GetDay`
   - `GetDate`

Label each intent with sample utterances

To help the model predict which intent a user is requesting, you must label each intent with some sample utterances.

1. In the pane on the left, select the **Data Labeling** page.

> **Tip**: You can expand the pane with the **>>** icon to see the page names, and hide it again with the **<<** icon.

1. Select the new **GetTime** intent and enter the utterance `what is the time?`. This adds the utterance as sample input for the intent.
2. Add the following additional utterances for the **GetTime** intent:
    - `what's the time?`
    - `what time is it?`
    - `tell me the time`

> **NOTE** To add a new utterance, write the utterance in the textbox next to the intent and then press ENTER.

3. Select the **GetDay** intent and add the following utterances as example input for that intent:
    - `what day is it?`
    - `what's the day?`
    - `what is the day today?`
    - `what day of the week is it?`
4. Select the **GetDate** intent and add the following utterances for it:
    - `what date is it?`
    - `what's the date?`
    - `what is the date today?`
    - `what's today's date?`
5. After you've added utterances for each of your intents, select **Save changes**.

Train and test the model

Now that you've added some intents, let's train the language model and see if it can correctly predict them from user input.

1. In the pane on the left, select **Training jobs**. Then select **+ Start a training job**.
2. On the **Start a training job** dialog, select the option to train a new model, name it `Clock`. Select **Standard training** mode and the default **Data splitting** options.
3. To begin the process of training your model, select **Train**.
4. When training is complete (which may take several minutes) the job **Status** will change to **Training succeeded**.
5. Select the **Model performance** page, and then select the **Clock** model. Review the overall and per-intent evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may be included in the results).

> **NOTE** To learn more about the evaluation metrics, refer to the documentation

6. Go to the **Deploying a model** page, then select **Add deployment**.

7. On the **Add deployment** dialog, select **Create a new deployment name**, and then enter `production`.
8. Select the **Clock** model in the **Model** field then select **Deploy**. The deployment may take some time.
9. When the model has been deployed, select the **Testing deployments** page, then select the **production** deployment in the **Deployment name** field.
10. Enter the following text in the empty textbox, and then select **Run the test**:

    ```
    what's the time now?
    ```

    Review the result that is returned, noting that it includes the predicted intent (which should be **GetTime**) and a confidence score that indicates the probability the model calculated for the predicted intent. The JSON tab shows the comparative confidence for each potential intent (the one with the highest confidence score is the predicted intent)

11. Clear the text box, and then run another test with the following text:

    ```
    tell me the time
    ```

    Again, review the predicted intent and confidence score.

12. Try the following text:

    ```
    what's the day today?
    ```

    Hopefully the model predicts the **GetDay** intent.

Add entities

So far you've defined some simple utterances that map to intents. Most real applications include more complex utterances from which specific data entities must be extracted to get more context for the intent.

Add a learned entity

The most common kind of entity is a *learned* entity, in which the model learns to identify entity values based on examples.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Location` and ensure that the **Learned** tab is selected. Then select **Add entity**.
3. After the **Location** entity has been created, return to the **Data labeling** page.
4. Select the **GetTime** intent and enter the following new example utterance:

    ```
    what time is it in London?
    ```

5. When the utterance has been added, select the word **London**, and in the drop-down list that appears, select **Location** to indicate that "London" is an example of a location.
6. Add another example utterance for the **GetTime** intent:

    ```
    Tell me the time in Paris?
    ```

7. When the utterance has been added, select the word **Paris**, and map it to the **Location** entity.
8. Add another example utterance for the **GetTime** intent:

    ```
    what's the time in New York?
    ```

9. When the utterance has been added, select the words **New York**, and map them to the **Location** entity.
10. Select **Save changes** to save the new utterances.

Add a *list* entity

In some cases, valid values for an entity can be restricted to a list of specific terms and synonyms; which can help the app identify instances of the entity in utterances.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Weekday` and select the **List** entity tab. Then select **Add entity**.
3. On the page for the **Weekday** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **List** section, select **+ Add new list**. Then enter the following value and synonym and select **Save**:

List key	synonyms
`Sunday`	`Sun`

> **NOTE** To enter the fields of the new list, insert the value `Sunday` in the text field, then click on the field where 'Type in value and press enter...' is displayed, enter the synonyms, and press ENTER.

4. Repeat the previous step to add the following list components:

Value	synonyms
`Monday`	`Mon`
`Tuesday`	`Tue, Tues`
`Wednesday`	`Wed, Weds`
`Thursday`	`Thur, Thurs`
`Friday`	`Fri`
`Saturday`	`Sat`

5. After adding and saving the list values, return to the **Data labeling** page.
6. Select the **GetDate** intent and enter the following new example utterance:

```
what date was it on Saturday?
```

7. When the utterance has been added, select the word ***Saturday***, and in the drop-down list that appears, select **Weekday**.
8. Add another example utterance for the **GetDate** intent:

```
what date will it be on Friday?
```

9. When the utterance has been added, map **Friday** to the **Weekday** entity.
10. Add another example utterance for the **GetDate** intent:

```
what will the date be on Thurs?
```

11. When the utterance has been added, map **Thurs** to the **Weekday** entity.
12. select **Save changes** to save the new utterances.

Add a *prebuilt* entity

The Azure AI Language service provides a set of *prebuilt* entities that are commonly used in conversational applications.

1. In Language Studio, return to the **Schema definition** page and then on the **Entities** tab, select **+ Add** to add a new entity.
2. In the **Add an entity** dialog box, enter the entity name `Date` and select the **Prebuilt** entity tab. Then select **Add entity**.
3. On the page for the **Date** entity, in the **Learned** section, ensure **Not required** is selected. Then, in the **Prebuilt** section, select **+ Add new prebuilt**.
4. In the **Select prebuilt** list, select **DateTime** and then select **Save**.
5. After adding the prebuilt entity, return to the **Data labeling** page
6. Select the **GetDay** intent and enter the following new example utterance:

   ```
   what day was 01/01/1901?
   ```

7. When the utterance has been added, select ***01/01/1901***, and in the drop-down list that appears, select **Date**.
8. Add another example utterance for the **GetDay** intent:

   ```
   what day will it be on Dec 31st 2099?
   ```

9. When the utterance has been added, map **Dec 31st 2099** to the **Date** entity.
10. Select **Save changes** to save the new utterances.

Retrain the model

Now that you've modified the schema, you need to retrain and retest the model.

1. On the **Training jobs** page, select **Start a training job**.
2. On the **Start a training job** dialog, select **overwrite an existing model** and specify the **Clock** model. Select **Train** to train the model. If prompted, confirm you want to overwrite the existing model.
3. When training is complete the job **Status** will update to **Training succeeded**.
4. Select the **Model performance** page and then select the **Clock** model. Review the evaluation metrics (*precision*, *recall*, and *F1 score*) and the *confusion matrix* generated by the evaluation that was performed when training (note that due to the small number of sample utterances, not all intents may be included in the results).
5. On the **Deploying a model** page, select **Add deployment**.
6. On the **Add deployment** dialog, select **Override an existing deployment name**, and then select **production**.
7. Select the **Clock** model in the **Model** field and then select **Deploy** to deploy it. This may take some time.
8. When the model is deployed, on the **Testing deployments** page, select the **production** deployment under the **Deployment name** field, and then test it with the following text:

   ```
   what's the time in Edinburgh?
   ```

9. Review the result that is returned, which should hopefully predict the **GetTime** intent and a **Location** entity with the text value "Edinburgh".
10. Try testing the following utterances:

    ```
    what time is it in Tokyo?
    ```

    ```
    what date is it on Friday?
    ```

    ```
    what's the date on Weds?
    ```

    ```
    what day was 01/01/2020?
    ```

```
what day will Mar 7th 2030 be?
```

Use the model from a client app

In a real project, you'd iteratively refine intents and entities, retrain, and retest until you are satisfied with the predictive performance. Then, when you've tested it and are satisfied with its predictive performance, you can use it in a client app by calling its REST interface or a runtime-specific SDK.

Prepare to develop an app in Cloud Shell

You'll develop your language understanding app using Cloud Shell in the Azure portal. The code files for your app have been provided in a GitHub repo.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-language/Labfiles/03-language/Python/clock-client
   ```

Configure your application

1. In the command line pane, run the following command to view the code files in the **clock-client** folder:

   Code Copy

   ```
   ls -a -l
   ```

   The files include a configuration file (**.env**) and a code file (**clock-client.py**).

2. Create a Python virtual environment and install the Azure AI Language Conversations SDK package and other required packages by running the following command:

   Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-language-conversations==1.1.0
```

3. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to the application

1. Enter the following command to edit the application code file:

Code Copy

```
code clock-client.py
```

2. Review the existing code. You will add code to work with the AI Language Conversations SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the AI Language Conversations SDK:

Code Copy

```
# Import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.language.conversations import ConversationAnalysisClient
```

4. In the **main** function, note that code to load the prediction endpoint and key from the configuration file has already been provided. Then find the comment **Create a client for the Language service model** and add the following code to create a conversation analysis client for your AI Language service:

Code Copy

```
# Create a client for the Language service model
client = ConversationAnalysisClient(
    ls_prediction_endpoint, AzureKeyCredential(ls_prediction_key))
```

5. Note that the code in the **main** function prompts for user input until the user enters "quit". Within this loop, find the comment **Call the Language service model to get intent and entities** and add the following code:

Code Copy

```
# Call the Language service model to get intent and entities
cls_project = 'Clock'
deployment_slot = 'production'

with client:
    query = userText
    result = client.analyze_conversation(
        task={
            "kind": "Conversation",
            "analysisInput": {
                "conversationItem": {
                    "participantId": "1",
                    "id": "1",
                    "modality": "text",
                    "language": "en",
                    "text": query
                },
                "isLoggingEnabled": False
            },
            "parameters": {
                "projectName": cls_project,
                "deploymentName": deployment_slot,
                "verbose": True
            }
        }
    )

top_intent = result["result"]["prediction"]["topIntent"]
entities = result["result"]["prediction"]["entities"]

print("view top intent:")
print("\ttop intent: {}".format(result["result"]["prediction"]["topIntent"]))
print("\tcategory: {}".format(result["result"]["prediction"]["intents"][0]["category"]))
print("\tconfidence score: {}\n".format(result["result"]["prediction"]["intents"][0]
["confidenceScore"]))

print("view entities:")
for entity in entities:
    print("\tcategory: {}".format(entity["category"]))
    print("\ttext: {}".format(entity["text"]))
    print("\tconfidence score: {}".format(entity["confidenceScore"]))

print("query: {}".format(result["result"]["query"]))
```

The call to the conversational understanding model returns a prediction/result, which includes the top (most likely) intent as well as any entities that were detected in the input utterance. Your client application must now use that prediction to determine and perform the appropriate action.

6. Find the comment **Apply the appropriate action**, and add the following code, which checks for intents supported by the application (**GetTime**, **GetDate**, and **GetDay**) and determines if any relevant entities have been detected, before calling an existing function to produce an appropriate response.

Code Copy

```
# Apply the appropriate action
if top_intent == 'GetTime':
    location = 'local'
    # Check for entities
    if len(entities) > 0:
        # Check for a location entity
        for entity in entities:
            if 'Location' == entity["category"]:
                # ML entities are strings, get the first one
                location = entity["text"]
    # Get the time for the specified location
    print(GetTime(location))

elif top_intent == 'GetDay':
    date_string = date.today().strftime("%m/%d/%Y")
    # Check for entities
    if len(entities) > 0:
        # Check for a Date entity
        for entity in entities:
            if 'Date' == entity["category"]:
                # Regex entities are strings, get the first one
                date_string = entity["text"]
    # Get the day for the specified date
    print(GetDay(date_string))

elif top_intent == 'GetDate':
    day = 'today'
    # Check for entities
    if len(entities) > 0:
        # Check for a Weekday entity
        for entity in entities:
            if 'Weekday' == entity["category"]:
                # List entities are lists
                day = entity["text"]
    # Get the date for the specified day
    print(GetDate(day))

else:
    # Some other intent (for example, "None") was predicted
    print('Try asking me for the time, the day, or the date.')
```

7. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python clock-client.py
```

8. When prompted, enter utterances to test the application. For example, try:

   Hello

   What time is it?

   What's the time in London?

   What's the date?

   What date is Sunday?

What day is it?

What day is 01/01/2025?

> **Note**: The logic in the application is deliberately simple, and has a number of limitations. For example, when getting the time, only a restricted set of cities is supported and daylight savings time is ignored. The goal is to see an example of a typical pattern for using Language Service in which your application must:
>
> a. Connect to a prediction endpoint.
> b. Submit an utterance to get a prediction.
> c. Implement logic to respond appropriately to the predicted intent and entities.

9. When you have finished testing, enter *quit*.

Clean up resources

If you're finished exploring the Azure AI Language service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Language resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

To learn more about conversational language understanding in Azure AI Language, see the Azure AI Language documentation.

Provision an Azure AI Language resource

Configure role-based access for your user

Upload sample articles

Create a custom text classification project

Label your data

Train your model

Evaluate your model

Deploy your model

Prepare to develop an app in Cloud Shell

Configure your application

Add code to classify documents

Clean up

Custom text classification

Azure AI Language provides several NLP capabilities, including the key phrase identification, text summarization, and sentiment analysis. The Language service also provides custom features like custom question answering and custom text classification.

To test the custom text classification of the Azure AI Language service, you'll configure the model using Language Studio then use a Python application to test it.

While this exercise is based on Python, you can develop text classification applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, use custom text classification, you need to enable the **Custom text classification & extraction** feature.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select **Create a resource**.
3. In the search field, search for **Language service**. Then, in the results, select **Create** under **Language Service**.
4. Select the box that includes **Custom text classification**. Then select **Continue to create your resource**.
5. Create a resource with the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Select or create a resource group*.
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North
        - UK South
        - West Europe
        - West US 2
        - West US 3
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Storage account**: New storage account
        - **Storage account name**: *Enter a unique name*.
        - **Storage account type**: Standard LRS
    - **Responsible AI notice**: Selected.
6. Select **Review + create,** then select **Create** to provision the resource.

7. Wait for deployment to complete, and then go to the resource group.

8. Find the storage account you created, select it, and verify the *Account kind* is **StorageV2**. If it's v1, upgrade your storage account kind on that resource page.

Configure role-based access for your user

> **NOTE**: If you skip this step, you'll get a 403 error when trying to connect to your custom project. It's important that your current user has this role to access storage account blob data, even if you're the owner of the storage account.

1. Go to your storage account page in the Azure portal.
2. Select **Access Control (IAM)** in the left navigation menu.
3. Select **Add** to Add Role Assignments, and choose the **Storage Blob Data Owner** role on the storage account.
4. Within **Assign access to**, select **User, group, or service principal**.
5. Select **Select members**.
6. Select your User. You can search for user names in the **Select** field.

Upload sample articles

Once you've created the Azure AI Language service and storage account, you'll need to upload example articles to train your model later.

1. In a new browser tab, download sample articles from `https://aka.ms/classification-articles` and extract the files to a folder of your choice.

2. In the Azure portal, navigate to the storage account you created, and select it.

3. In your storage account select **Configuration**, located below **Settings**. In the Configuration screen enable the option to **Allow Blob anonymous access** then select **Save**.

4. Select **Containers** in the left menu, located below **Data storage**. On the screen that appears, select **+ Container**. Give the container the name `articles`, and set **Anonymous access level** to **Container (anonymous read access for containers and blobs)**.

   > **NOTE**: When you configure a storage account for a real solution, be careful to assign the appropriate access level. To learn more about each access level, see the Azure Storage documentation.

5. After you've created the container, select it then select the **Upload** button. Select **Browse for files** to browse for the sample articles you downloaded. Then select **Upload**.

Create a custom text classification project

After configuration is complete, create a custom text classification project. This project provides a working place to build, train, and deploy your model.

> **NOTE**: This lab utilizes **Language Studio**, but you can also create, build, train, and deploy your model through the REST API.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.

2. If prompted to choose a Language resource, select the following settings:

   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.

- **Language resource**: The Azure AI Language resource you created previously.

If you are <u>not</u> prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

a. On the bar at the top if the page, select the **Settings (⚙)** button.
b. On the **Settings** page, view the **Resources** tab.
c. Select the language resource you just created, and click **Switch resource**.
d. At the top of the page, click **Language Studio** to return to the Language Studio home page

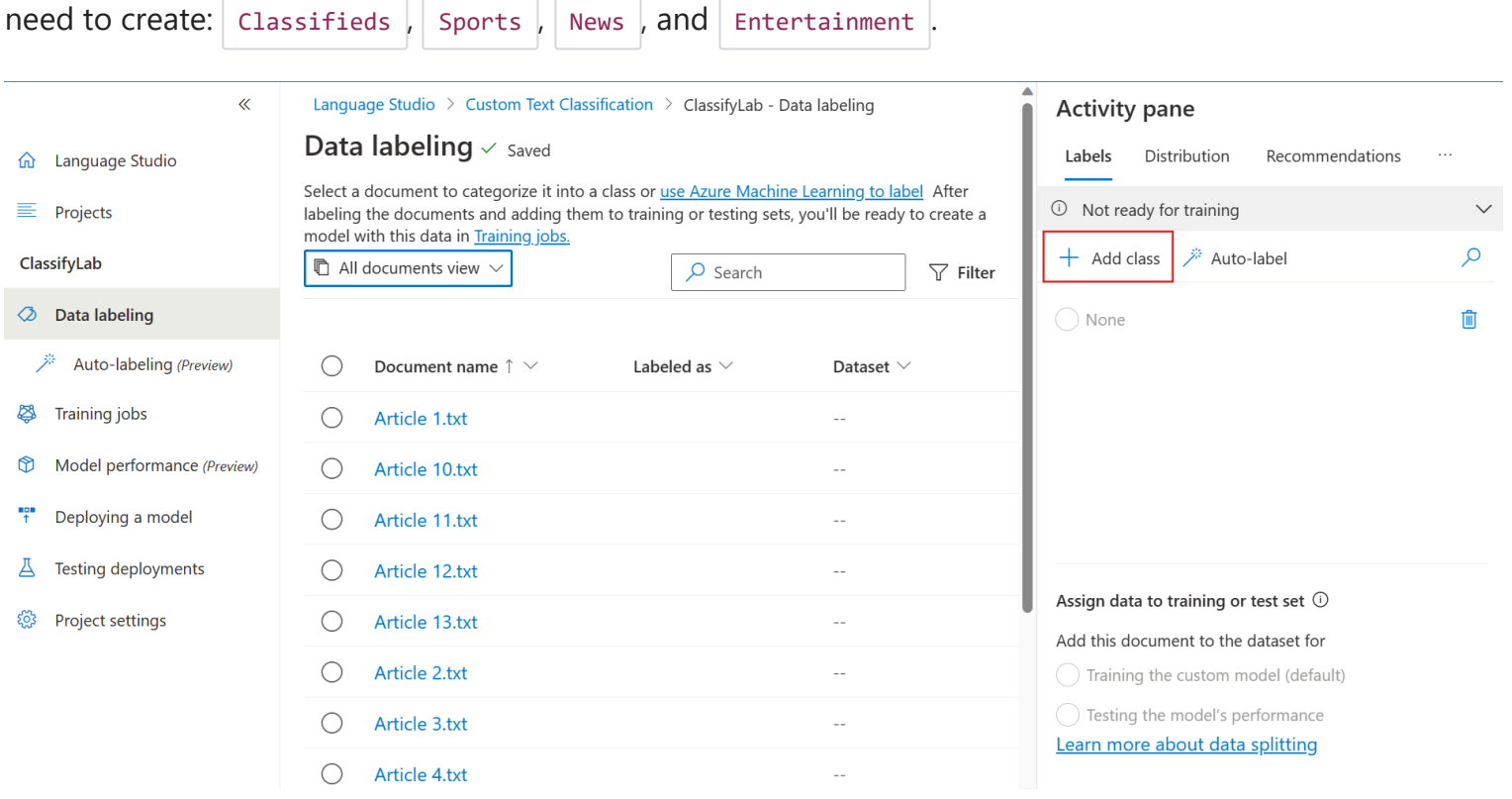
3. At the top of the portal, in the **Create new** menu, select **Custom text classification**.
4. The **Connect storage** page appears. All values will already have been filled. So select **Next**.
5. On the **Select project type** page, select **Single label classification**. Then select **Next**.
6. On the **Enter basic information** pane, set the following:
   - **Name**: `ClassifyLab`
   - **Text primary language**: English (US)
   - **Description**: `Custom text lab`
7. Select **Next**.
8. On the **Choose container** page, set the **Blob store container** dropdown to your *articles* container.
9. Select the **No, I need to label my files as part of this project** option. Then select **Next**.
10. Select **Create project**.

> **Tip**: If you get an error about not being authorized to perform this operation, you'll need to add a role assignment. To fix this, we add the role "Storage Blob Data Contributor" on the storage account for the user running the lab. More details can be found on the documentation page

Label your data

Now that your project is created, you need to label, or tag, your data to train your model how to classify text.

1. On the left, select **Data labeling**, if not already selected. You'll see a list of the files you uploaded to your storage account.
2. On the right side, in the **Activity** pane, select **+ Add class**. The articles in this lab fall into four classes you'll need to create: `Classifieds`, `Sports`, `News`, and `Entertainment`.



3. After you've created your four classes, select **Article 1** to start. Here you can read the article, define which class this file is, and which dataset (training or testing) to assign it to.

4. Assign each article the appropriate class and dataset (training or testing) using the **Activity** pane on the right. You can select a label from the list of labels on the right, and set each article to **training** or **testing** using the options at the bottom of the Activity pane. You select **Next document** to move to the next document. For the purposes of this lab, we'll define which are to be used for training the model and testing the model:

Article	Class	Dataset
Article 1	Sports	Training
Article 10	News	Training
Article 11	Entertainment	Testing
Article 12	News	Testing
Article 13	Sports	Testing
Article 2	Sports	Training
Article 3	Classifieds	Training
Article 4	Classifieds	Training
Article 5	Entertainment	Training
Article 6	Entertainment	Training
Article 7	News	Training
Article 8	News	Training
Article 9	Entertainment	Training

> **NOTE** Files in Language Studio are listed alphabetically, which is why the above list is not in sequential order. Make sure you visit both pages of documents when labeling your articles.

5. Select **Save labels** to save your labels.

Train your model

After you've labeled your data, you need to train your model.

1. Select **Training jobs** on the left side menu.
2. Select **Start a training job**.
3. Train a new model named `ClassifyArticles`.
4. Select **Use a manual split of training and testing data**.

> **TIP** In your own classification projects, the Azure AI Language service will automatically split the testing set by percentage which is useful with a large dataset. With smaller datasets, it's important to train with the right class distribution.

5. Select **Train**

> **IMPORTANT** Training your model can sometimes take several minutes. You'll get a notification when it's complete.

Evaluate your model

In real world applications of text classification, it's important to evaluate and improve your model to verify it's performing as you expect.

1. Select **Model performance**, and select your **ClassifyArticles** model. There you can see the scoring of your model, performance metrics, and when it was trained. If the scoring of your model isn't 100%, it means that one of the documents used for testing didn't evaluate to what it was labeled. These failures can help you understand where to improve.
2. Select **Test set details** tab. If there are any errors, this tab allows you to see the articles you indicated for testing and what the model predicted them as and whether that conflicts with their test label. The tab defaults to show incorrect predictions only. You can toggle the **Show mismatches only** option to see all the articles you indicated for testing and what they each of them predicted as.

Deploy your model

When you're satisfied with the training of your model, it's time to deploy it, which allows you to start classifying text through the API.

1. On the left panel, select **Deploying model**.
2. Select **Add deployment**, then enter `articles` in the **Create a new deployment name** field, and select **ClassifyArticles** in the **Model** field.
3. Select **Deploy** to deploy your model.
4. Once your model is deployed, leave that page open. You'll need your project and deployment name in the next step.

Prepare to develop an app in Cloud Shell

To test the custom text classification capabilities of the Azure AI Language service, you'll develop a simple console application in the Azure Cloud Shell.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

```
cd mslearn-ai-language/Labfiles/04-text-classification/Python/classify-text
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **classify-text** folder:

   Code Copy

   ```
   ls -a -l
   ```

   The files include a configuration file (**.env**) and a code file (**classify-text.py**). The text your application will analyze is in the **articles** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-ai-textanalytics==5.3.0
   ```

3. Enter the following command to edit the application configuration file:

   Code Copy

   ```
   code .env
   ```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).The file should already contain the project and deployment names for your text classification model.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to classify documents

1. Enter the following command to edit the application code file:

   Code Copy

   ```
   code classify-text.py
   ```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

   > **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

   Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key and the project and deployment names from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a text analysis client:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Note that the existing code reads all of the files in the **articles** folder and creates a list containing their contents. Then find the comment **Get Classifications** and add the following code:

Code Copy

```
# Get Classifications
operation = ai_client.begin_single_label_classify(
    batchedDocuments,
    project_name=project_name,
    deployment_name=deployment_name
)

document_results = operation.result()

for doc, classification_result in zip(files, document_results):
    if classification_result.kind == "CustomDocumentClassification":
        classification = classification_result.classifications[0]
        print("{} was classified as '{}' with confidence score {}.".format(
            doc, classification.category, classification.confidence_score)
        )
    elif classification_result.is_error is True:
        print("{} has an error with code '{}' and message '{}'".format(
            doc, classification_result.error.code, classification_result.error.message)
        )
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python classify-text.py
```

7. Observe the output. The application should list a classification and confidence score for each text file.

Clean up

When you don't need your project any more, you can delete if from your **Projects** page in Language Studio. You can also remove the Azure AI Language service and associated storage account in the Azure portal.

Provision an Azure AI Language resource

Configure role-based access for your user

Upload sample ads

Create a custom named entity recognition project

Label your data

Train your model

Evaluate your model

Deploy your model

Prepare to develop an app in Cloud Shell

Configure your application

Add code to extract entities

Clean up

Extract custom entities

In addition to other natural language processing capabilities, Azure AI Language Service enables you to define custom entities, and extract instances of them from text.

To test the custom entity extraction, we'll create a model and train it through Azure AI Language Studio, then use a Python application to test it.

While this exercise is based on Python, you can develop text classification applications using multiple language-specific SDKs; including:

- Azure AI Text Analytics client library for Python
- Azure AI Text Analytics client library for .NET
- Azure AI Text Analytics client library for JavaScript

This exercise takes approximately **35** minutes.

Provision an *Azure AI Language* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Language service** resource. Additionally, use custom text classification, you need to enable the **Custom text classification & extraction** feature.

1. In a browser, open the Azure portal at `https://portal.azure.com`, and sign in with your Microsoft account.
2. Select the **Create a resource** button, search for *Language*, and create a **Language Service** resource. When on the page for *Select additional features*, select the custom feature containing **Custom named entity recognition extraction**. Create the resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Select or create a resource group*
    - **Region**: *Choose from one of the following regions**
        - Australia East
        - Central India
        - East US
        - East US 2
        - North Europe
        - South Central US
        - Switzerland North
        - UK South
        - West Europe
        - West US 2
        - West US 3
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
    - **Storage account**: New storage account:
        - **Storage account name**: *Enter a unique name*.
        - **Storage account type**: Standard LRS
    - **Responsible AI notice**: Selected.
3. Select **Review + create**, then select **Create** to provision the resource.
4. Wait for deployment to complete, and then go to the deployed resource.
5. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Configure role-based access for your user

> **NOTE**: If you skip this step, you'll have a 403 error when trying to connect to your custom project. It's important that your current user has this role to access storage account blob data, even if you're the owner of the storage account.

1. Go to your storage account page in the Azure portal.
2. Select **Access Control (IAM)** in the left navigation menu.
3. Select **Add** to Add Role Assignments, and choose the **Storage Blob Data Contributor** role on the storage account.
4. Within **Assign access to**, select **User, group, or service principal**.
5. Select **Select members**.
6. Select your User. You can search for user names in the **Select** field.

Upload sample ads

After you've created the Azure AI Language Service and storage account, you'll need to upload example ads to train your model later.

1. In a new browser tab, download sample classified ads from `https://aka.ms/entity-extraction-ads` and extract the files to a folder of your choice.
2. In the Azure portal, navigate to the storage account you created, and select it.
3. In your storage account select **Configuration**, located below **Settings**, and screen enable the option to **Allow Blob anonymous access** then select **Save**.
4. Select **Containers** from the left menu, located below **Data storage**. On the screen that appears, select **+ Container**. Give the container the name `classifieds`, and set **Anonymous access level** to **Container (anonymous read access for containers and blobs)**.

   > **NOTE**: When you configure a storage account for a real solution, be careful to assign the appropriate access level. To learn more about each access level, see the Azure Storage documentation.

5. After creating the container, select it and click the **Upload** button and upload the sample ads you downloaded.

Create a custom named entity recognition project

Now you're ready to create a custom named entity recognition project. This project provides a working place to build, train, and deploy your model.

> **NOTE**: You can also create, build, train, and deploy your model through the REST API.

1. In a new browser tab, open the Azure AI Language Studio portal at `https://language.cognitive.azure.com/` and sign in using the Microsoft account associated with your Azure subscription.
2. If prompted to choose a Language resource, select the following settings:
   - **Azure Directory**: The Azure directory containing your subscription.
   - **Azure subscription**: Your Azure subscription.
   - **Resource type**: Language.
   - **Language resource**: The Azure AI Language resource you created previously.

   If you are not prompted to choose a language resource, it may be because you have multiple Language resources in your subscription; in which case:

   a. On the bar at the top of the page, select the **Settings (⚙)** button.
   b. On the **Settings** page, view the **Resources** tab.

c. Select the language resource you just created, and click **Switch resource**.
d. At the top of the page, click **Language Studio** to return to the Language Studio home page.

3. At the top of the portal, in the **Create new** menu, select **Custom named entity recognition**.

4. Create a new project with the following settings:

- **Connect storage**: *This value is likely already filled. Change it to your storage account if it isn't already*
- **Basic information**:
- **Name**: `CustomEntityLab`
    - **Text primary language**: English (US)
    - **Does your dataset include documents that are not in the same language?** : *No*
    - **Description**: `Custom entities in classified ads`
- **Container**:
    - **Blob store container**: classifieds
    - **Are your files labeled with classes?**: No, I need to label my files as part of this project

> **Tip**: If you get an error about not being authorized to perform this operation, you'll need to add a role assignment. To fix this, we add the role "Storage Blob Data Contributor" on the storage account for the user running the lab. More details can be found on the documentation page

Label your data

Now that your project is created, you need to label your data to train your model how to identity entities.

1. If the **Data labeling** page is not already open, in the pane on the left, select **Data labeling**. You'll see a list of the files you uploaded to your storage account.
2. On the right side, in the **Activity** pane, select **Add entity** and add a new entity named `ItemForSale`.
3. Repeat the previous step to create the following entities:
    - `Price`
    - `Location`
4. After you've created your three entities, select **Ad 1.txt** so you can read it.
5. In *Ad 1.txt*:
    a. Highlight the text *face cord of firewood* and select the **ItemForSale** entity.
    b. Highlight the text *Denver, CO* and select the **Location** entity.
    c. Highlight the text *$90* and select the **Price** entity.
6. In the **Activity** pane, note that this document will be added to the dataset for training the model.
7. Use the **Next document** button to move to the next document, and continue assigning text to appropriate entities for the entire set of documents, adding them all to the training dataset.
8. When you have labeled the last document (*Ad 9.txt*), save the labels.

Train your model

After you've labeled your data, you need to train your model.

1. Select **Training jobs** in the pane on the left.
2. Select **Start a training job**
3. Train a new model named `ExtractAds`
4. Choose **Automatically split the testing set from training data**

> **TIP**: In your own extraction projects, use the testing split that best suits your data. For more consistent data and larger datasets, the Azure AI Language Service will automatically split the testing set by percentage. With smaller datasets, it's important to train with the right variety of possible input documents.

5. Click **Train**

> **IMPORTANT**: Training your model can sometimes take several minutes. You'll get a notification when it's complete.

Evaluate your model

In real world applications, it's important to evaluate and improve your model to verify it's performing as you expect. Two pages on the left show you the details of your trained model, and any testing that failed.

Select **Model performance** on the left side menu, and select your `ExtractAds` model. There you can see the scoring of your model, performance metrics, and when it was trained. You'll be able to see if any testing documents failed, and these failures help you understand where to improve.

Deploy your model

When you're satisfied with the training of your model, it's time to deploy it, which allows you to start extracting entities through the API.

1. In the left pane, select **Deploying a model**.
2. Select **Add deployment**, then enter the name `AdEntities` and select the **ExtractAds** model.
3. Click **Deploy** to deploy your model.

Prepare to develop an app in Cloud Shell

To test the custom entity extraction capabilities of the Azure AI Language service, you'll develop a simple console application in the Azure Cloud Shell.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

   Code Copy

   ```
   rm -r mslearn-ai-language -f
   git clone https://github.com/microsoftlearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

   Code Copy

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/05-custom-entity-recognition/Python/custom-entities
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **custom-entities** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**custom-entities.py**). The text your application will analyze is in the **ads** subfolder.

2. Create a Python virtual environment and install the Azure AI Language Text Analytics SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-textanalytics==5.3.0
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **endpoint** and a **key** from the Azure Language resource you created (available on the **Keys and Endpoint** page for your Azure AI Language resource in the Azure portal).The file should already contain the project and deployment names for your custom entity extraction model.

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to extract entities

1. Enter the following command to edit the application code file:

```
code custom-entities.py
```

2. Review the existing code. You will add code to work with the AI Language Text Analytics SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Text Analytics SDK:

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.textanalytics import TextAnalyticsClient
```

4. In the **main** function, note that code to load the Azure AI Language service endpoint and key and the project and deployment names from the configuration file has already been provided. Then find the comment **Create client using endpoint and key**, and add the following code to create a text analytics client:

```
# Create client using endpoint and key
credential = AzureKeyCredential(ai_key)
ai_client = TextAnalyticsClient(endpoint=ai_endpoint, credential=credential)
```

5. Note that the existing code reads all of the files in the **ads** folder and creates a list containing their contents. Then find the comment **Extract entities** and add the following code:

```
# Extract entities
operation = ai_client.begin_recognize_custom_entities(
    batchedDocuments,
    project_name=project_name,
    deployment_name=deployment_name
)

document_results = operation.result()

for doc, custom_entities_result in zip(files, document_results):
    print(doc)
    if custom_entities_result.kind == "CustomEntityRecognition":
        for entity in custom_entities_result.entities:
            print(
                "\tEntity '{}' has category '{}' with confidence score of '{}'".format(
                    entity.text, entity.category, entity.confidence_score
                )
            )
    elif custom_entities_result.is_error is True:
        print("\tError with code '{}' and message '{}'".format(
            custom_entities_result.error.code, custom_entities_result.error.message
            )
        )
```

6. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

```
python custom-entities.py
```

7. Observe the output. The application should list details of the entities found in each text file.

Clean up

When you don't need your project anymore, you can delete if from your **Projects** page in Language Studio. You can also remove the Azure AI Language service and associated storage account in the Azure portal.

Provision an Azure AI Translator resource

Prepare to develop an app in Cloud Shell

Configure your application

Add code to translate text

Clean up resources

More information

Translate Text

Azure AI Translator is a service that enables you to translate text between languages. In this exercise, you'll use it to create a simple app that translates input in any supported language to the target language of your choice.

While this exercise is based on Python, you can develop text translation applications using multiple language-specific SDKs; including:

- Azure AI Translation client library for Python
- Azure AI Translation client library for .NET
- Azure AI Translation client library for JavaScript

This exercise takes approximately **30** minutes.

Provision an *Azure AI Translator* resource

If you don't already have one in your subscription, you'll need to provision an **Azure AI Translator** resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the search field at the top, search for **Translators** then select **Translators** in the results.
3. Create a resource with the following settings:
    - **Subscription**: *Your Azure subscription*
    - **Resource group**: *Choose or create a resource group*
    - **Region**: *Choose any available region*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page. You will need the information on this page later in the exercise.

Prepare to develop an app in Cloud Shell

To test the text translation capabilities of Azure AI Translator, you'll develop a simple console application in the Azure Cloud Shell.

1. In the Azure Portal, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

    > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the application code files:

```
cd mslearn-ai-language/Labfiles/06-translator-sdk/Python/translate-text
```

Configure your application

1. In the command line pane, run the following command to view the code files in the **translate-text** folder:

```
ls -a -l
```

   The files include a configuration file (**.env**) and a code file (**translate.py**).

2. Create a Python virtual environment and install the Azure AI Translation SDK package and other required packages by running the following command:

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-ai-translation-text==1.0.1
```

3. Enter the following command to edit the application configuration file:

```
code .env
```

   The file is opened in a code editor.

4. Update the configuration values to include the **region** and a **key** from the Azure AI Translator resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

> **NOTE**: Be sure to add the *region* for your resource, not the endpoint!

5. After you've replaced the placeholders, within the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Add code to translate text

1. Enter the following command to edit the application code file:

```
code translate.py
```

2. Review the existing code. You will add code to work with the Azure AI Translation SDK.

> **Tip**: As you add code to the code file, be sure to maintain the correct indentation.

3. At the top of the code file, under the existing namespace references, find the comment **Import namespaces** and add the following code to import the namespaces you will need to use the Translation SDK:

Code Copy

```
# import namespaces
from azure.core.credentials import AzureKeyCredential
from azure.ai.translation.text import *
from azure.ai.translation.text.models import InputTextItem
```

4. In the **main** function, note that the existing code reads the configuration settings.

5. Find the comment **Create client using endpoint and key** and add the following code:

Code Copy

```
# Create client using endpoint and key
credential = AzureKeyCredential(translatorKey)
client = TextTranslationClient(credential=credential, region=translatorRegion)
```

6. Find the comment **Choose target language** and add the following code, which uses the Text Translator service to return list of supported languages for translation, and prompts the user to select a language code for the target language:

Code Copy

```
# Choose target language
languagesResponse = client.get_supported_languages(scope="translation")
print("{} languages supported.".format(len(languagesResponse.translation)))
print("(See https://learn.microsoft.com/azure/ai-services/translator/language-support#translation)")
print("Enter a target language code for translation (for example, 'en'):")
targetLanguage = "xx"
supportedLanguage = False
while supportedLanguage == False:
    targetLanguage = input()
    if  targetLanguage in languagesResponse.translation.keys():
        supportedLanguage = True
    else:
        print("{} is not a supported language.".format(targetLanguage))
```

7. Find the comment **Translate text** and add the following code, which repeatedly prompts the user for text to be translated, uses the Azure AI Translator service to translate it to the target language (detecting the source language automatically), and displays the results until the user enters *quit*:

Code Copy

```
# Translate text
inputText = ""
while inputText.lower() != "quit":
    inputText = input("Enter text to translate ('quit' to exit):")
    if inputText != "quit":
        input_text_elements = [InputTextItem(text=inputText)]
        translationResponse = client.translate(body=input_text_elements, to_language=
[targetLanguage])
        translation = translationResponse[0] if translationResponse else None
        if translation:
            sourceLanguage = translation.detected_language
            for translated_text in translation.translations:
                print(f"'{inputText}' was translated from {sourceLanguage.language} to
{translated_text.to} as '{translated_text.text}'.")
```

8. Save your changes (CTRL+S), then enter the following command to run the program (you maximize the cloud shell pane and resize the panels to see more text in the command line pane):

Code Copy

```
python translate.py
```

9. When prompted, enter a valid target language from the list displayed.
10. Enter a phrase to be translated (for example `This is a test` or `C'est un test`) and view the results, which should detect the source language and translate the text to the target language.
11. When you're done, enter `quit`. You can run the application again and choose a different target language.

Clean up resources

If you're finished exploring the Azure AI Translator service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Translator resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

More information

For more information about using **Azure AI Translator**, see the Azure AI Translator documentation.

Create an Azure AI Speech resource

Prepare and configure the speaking clock app

Add code to use the Azure AI Speech SDK

Run the app

Add code to recognize speech

Synthesize speech

Use Speech Synthesis Markup Language

Clean up

What if you have a mic and speaker?

More information

Recognize and synthesize speech

Azure AI Speech is a service that provides speech-related functionality, including:

- A *speech-to-text* API that enables you to implement speech recognition (converting audible spoken words into text).
- A *text-to-speech* API that enables you to implement speech synthesis (converting text into audible speech).

In this exercise, you'll use both of these APIs to implement a speaking clock application.

While this exercise is based on Python, you can develop speech applications using multiple language-specific SDKs; including:

- Azure AI Speech SDK for Python
- Azure AI Speech SDK for .NET
- Azure AI Speech SDK for JavaScript

This exercise takes approximately **30** minutes.

> **NOTE** This exercise is designed to be completed in the Azure cloud shell, where direct access to your computer's sound hardware is not supported. The lab will therefore use audio files for speech input and output streams. The code to achieve the same results using a mic and speaker is provided for your reference.

Create an Azure AI Speech resource

Let's start by creating an Azure AI Speech resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search field, search for **Speech service**. Select it from the list, then select **Create**.
3. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Prepare and configure the speaking clock app

1. Leaving the **Keys and Endpoint** page open, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

    > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

Code Copy

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the speaking clock application code files:

Code Copy

```
cd mslearn-ai-language/Labfiles/07-speech/Python/speaking-clock
```

5. In the command line pane, run the following command to view the code files in the **speaking-clock** folder:

Code Copy

```
ls -a -l
```

The files include a configuration file (**.env**) and a code file (**speaking-clock.py**). The audio files your application will use are in the **audio** subfolder.

6. Create a Python virtual environment and install the Azure AI Speech SDK package and other required packages by running the following command:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-speech==1.42.0
```

7. Enter the following command to edit the configuration file:

Code Copy

```
code .env
```

The file is opened in a code editor.

8. Update the configuration values to include the **region** and a **key** from the Azure AI Speech resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to use the Azure AI Speech SDK

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code speaking-clock.py
```

2. At the top of the code file, under the existing namespace references, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Azure AI Speech SDK:

Code Copy

```
# Import namespaces
from azure.core.credentials import AzureKeyCredential
import azure.cognitiveservices.speech as speech_sdk
```

3. In the **main** function, under the comment **Get config settings**, note that the code loads the key and region you defined in the configuration file.

4. Find the comment **Configure speech service**, and add the following code to use the AI Services key and your region to configure your connection to the Azure AI Services Speech endpoint:

Code Copy

```
# Configure speech service
speech_config = speech_sdk.SpeechConfig(speech_key, speech_region)
print('Ready to use speech service in:', speech_config.region)
```

5. Save your changes (*CTRL+S*), but leave the code editor open.

Run the app

So far, the app doesn't do anything other than connect to your Azure AI Speech service, but it's useful to run it and check that it works before adding speech functionality.

1. In the command line, enter the following command to run the speaking clock app:

Code Copy

```
python speaking-clock.py
```

The code should display the region of the speech service resource the application will use. A successful run indicates that the app has connected to your Azure AI Speech resource.

Add code to recognize speech

Now that you have a **SpeechConfig** for the speech service in your project's Azure AI Services resource, you can use the **Speech-to-text** API to recognize speech and transcribe it to text.

In this procedure, the speech input is captured from an audio file, which you can play here:

1. In the code file, note that the code uses the **TranscribeCommand** function to accept spoken input. Then in the **TranscribeCommand** function, find the comment **Configure speech recognition** and add the appropriate code below to create a **SpeechRecognizer** client that can be used to recognize and transcribe speech from an audio file:

Code Copy

```
# Configure speech recognition
current_dir = os.getcwd()
audioFile = current_dir + '/time.wav'
audio_config = speech_sdk.AudioConfig(filename=audioFile)
speech_recognizer = speech_sdk.SpeechRecognizer(speech_config, audio_config)
```

2. In the **TranscribeCommand** function, under the comment **Process speech input**, add the following code to listen for spoken input, being careful not to replace the code at the end of the function that returns the command:

Code Copy

```
# Process speech input
print("Listening...")
speech = speech_recognizer.recognize_once_async().get()
if speech.reason == speech_sdk.ResultReason.RecognizedSpeech:
    command = speech.text
    print(command)
else:
    print(speech.reason)
    if speech.reason == speech_sdk.ResultReason.Canceled:
        cancellation = speech.cancellation_details
        print(cancellation.reason)
        print(cancellation.error_details)
```

3. Save your changes (*CTRL+S*), and then in the command line below the code editor, re-run the program:
4. Review the output, which should successfully "hear" the speech in the audio file and return an appropriate response (note that your Azure cloud shell may be running on a server that is in a different time-zone to yours!)

> **Tip**: If the SpeechRecognizer encounters an error, it produces a result of "Cancelled". The code in the application will then display the error message. The most likely cause is an incorrect region value in the configuration file.

Synthesize speech

Your speaking clock application accepts spoken input, but it doesn't actually speak! Let's fix that by adding code to synthesize speech.

Once again, due to the hardware limitations of the cloud shell we'll direct the synthesized speech output to a file.

1. In the code file, note that the code uses the **TellTime** function to tell the user the current time.
2. In the **TellTime** function, under the comment **Configure speech synthesis**, add the following code to create a **SpeechSynthesizer** client that can be used to generate spoken output:

Code Copy

```
# Configure speech synthesis
output_file = "output.wav"
speech_config.speech_synthesis_voice_name = "en-GB-RyanNeural"
audio_config = speech_sdk.audio.AudioConfig(filename=output_file)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config,)
```

3. In the **TellTime** function, under the comment **Synthesize spoken output**, add the following code to generate spoken output, being careful not to replace the code at the end of the function that prints the response:

```
# Synthesize spoken output
speak = speech_synthesizer.speak_text_async(response_text).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
else:
    print("Spoken output saved in " + output_file)
```

4. Save your changes (*CTRL+S*) and re-run the program, which should indicate that the spoken output was saved in a file.

5. If you have a media player capable of playing .wav audio files, download the file that was generated by entering the following command:

```
download ./output.wav
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file.

The file should sound similar to this:

Use Speech Synthesis Markup Language

Speech Synthesis Markup Language (SSML) enables you to customize the way your speech is synthesized using an XML-based format.

1. In the **TellTime** function, replace all of the current code under the comment **Synthesize spoken output** with the following code (leave the code under the comment **Print the response**):

```
# Synthesize spoken output
responseSsml = " \
   <speak version='1.0' xmlns='http://www.w3.org/2001/10/synthesis' xml:lang='en-US'> \
       <voice name='en-GB-LibbyNeural'> \
           {} \
           <break strength='weak'/> \
           Time to end this lab! \
       </voice> \
   </speak>".format(response_text)
speak = speech_synthesizer.speak_ssml_async(responseSsml).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
   print(speak.reason)
else:
   print("Spoken output saved in " + output_file)
```

2. Save your changes and re-run the program, which should once again indicate that the spoken output was saved in a file.
3. Download and play the generated file, which should sound similar to this:

Clean up

If you've finished exploring Azure AI Speech, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Speech resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

What if you have a mic and speaker?

In this exercise, the Azure Cloud Shell environment we used doesn't support audio hardware, so you used audio files for the speech input and output. Let's see how the code can be modified to use audio hardware if you have it available.

Using speech recognition with a microphone

If you have a mic, you can use the following code to capture spoken input for speech recognition:

Code Copy

```
# Configure speech recognition
audio_config = speech_sdk.AudioConfig(use_default_microphone=True)
speech_recognizer = speech_sdk.SpeechRecognizer(speech_config, audio_config)
print('Speak now...')

# Process speech input
speech = speech_recognizer.recognize_once_async().get()
if speech.reason == speech_sdk.ResultReason.RecognizedSpeech:
    command = speech.text
    print(command)
else:
    print(speech.reason)
    if speech.reason == speech_sdk.ResultReason.Canceled:
        cancellation = speech.cancellation_details
        print(cancellation.reason)
        print(cancellation.error_details)
```

> **Note**: The system default microphone is the default audio input, so you could also just omit the AudioConfig altogether!

Using speech synthesis with a speaker

If you have a speaker, you can use the following code to synthesize speech.

Code Copy

```
response_text = 'The time is {}:{:02d}'.format(now.hour,now.minute)

# Configure speech synthesis
speech_config.speech_synthesis_voice_name = "en-GB-RyanNeural"
audio_config = speech_sdk.audio.AudioOutputConfig(use_default_speaker=True)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config)

# Synthesize spoken output
speak = speech_synthesizer.speak_text_async(response_text).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
```

> **Note**: The system default speaker is the default audio output, so you could also just omit the AudioConfig altogether!

More information

For more information about using the **Speech-to-text** and **Text-to-speech** APIs, see the Speech-to-text documentation and Text-to-speech documentation.

Create an Azure AI Speech resource

Prepare to develop an app in Cloud Shell

Add code to use the Azure AI Speech SDK

Run the app

Implement speech translation

Synthesize the translation to speech

Clean up resources

What if you have a mic and speaker?

More information

Translate Speech

Azure AI Speech includes a speech translation API that you can use to translate spoken language. For example, suppose you want to develop a translator application that people can use when traveling in places where they don't speak the local language. They would be able to say phrases such as "Where is the station?" or "I need to find a pharmacy" in their own language, and have it translate them to the local language. In this exercise, you'll use the Azure AI Speech SDK for Python to create a simple application based on this example.

While this exercise is based on Python, you can develop speech translation applications using multiple language-specific SDKs; including:

- Azure AI Speech SDK for Python
- Azure AI Speech SDK for .NET
- Azure AI Speech SDK for JavaScript

This exercise takes approximately **30** minutes.

> **NOTE** This exercise is designed to be completed in the Azure cloud shell, where direct access to your computer's sound hardware is not supported. The lab will therefore use audio files for speech input and output streams. The code to achieve the same results using a mic and speaker is provided for your reference.

Create an Azure AI Speech resource

Let's start by creating an Azure AI Speech resource.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. In the top search field, search for **Speech service**. Select it from the list, then select **Create**.
3. Provision the resource using the following settings:
    - **Subscription**: *Your Azure subscription*.
    - **Resource group**: *Choose or create a resource group*.
    - **Region**:*Choose any available region*
    - **Name**: *Enter a unique name*.
    - **Pricing tier**: Select **F0** (*free*), or **S** (*standard*) if F is not available.
4. Select **Review + create**, then select **Create** to provision the resource.
5. Wait for deployment to complete, and then go to the deployed resource.
6. View the **Keys and Endpoint** page in the **Resource Management** section. You will need the information on this page later in the exercise.

Prepare to develop an app in Cloud Shell

1. Leaving the **Keys and Endpoint** page open, use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

    > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

2. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

    Ensure you've switched to the classic version of the cloud shell before continuing.

3. In the PowerShell pane, enter the following commands to clone the GitHub repo for this exercise:

    Code Copy

```
rm -r mslearn-ai-language -f
git clone https://github.com/microsoftlearning/mslearn-ai-language
```

> **Tip**: As you enter commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

4. After the repo has been cloned, navigate to the folder containing the code files:

Code Copy

```
cd mslearn-ai-language/Labfiles/08-speech-translation/Python/translator
```

5. In the command line pane, run the following command to view the code files in the **translator** folder:

Code Copy

```
ls -a -l
```

The files include a configuration file (**.env**) and a code file (**translator.py**).

6. Create a Python virtual environment and install the Azure AI Speech SDK package and other required packages by running the following command:

Code Copy

```
python -m venv labenv
./labenv/bin/Activate.ps1
pip install -r requirements.txt azure-cognitiveservices-speech==1.42.0
```

7. Enter the following command to edit the configuration file that has been provided:

Code Copy

```
code .env
```

The file is opened in a code editor.

8. Update the configuration values to include the **region** and a **key** from the Azure AI Speech resource you created (available on the **Keys and Endpoint** page for your Azure AI Translator resource in the Azure portal).

9. After you've replaced the placeholders, use the **CTRL+S** command to save your changes and then use the **CTRL+Q** command to close the code editor while keeping the cloud shell command line open.

Add code to use the Azure AI Speech SDK

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file that has been provided:

Code Copy

```
code translator.py
```

2. At the top of the code file, under the existing namespace references, find the comment **Import namespaces**. Then, under this comment, add the following language-specific code to import the namespaces you will need to use the Azure AI Speech SDK:

```
Code
Copy

# Import namespaces
from azure.core.credentials import AzureKeyCredential
import azure.cognitiveservices.speech as speech_sdk
```

3. In the **main** function, under the comment **Get config settings**, note that the code loads the key and region you defined in the configuration file.

4. Find the following code under the comment **Configure translation**, and add the following code to configure your connection to the Azure AI Services Speech endpoint:

```
Code
Copy

# Configure translation
translation_config = speech_sdk.translation.SpeechTranslationConfig(speech_key,
speech_region)
translation_config.speech_recognition_language = 'en-US'
translation_config.add_target_language('fr')
translation_config.add_target_language('es')
translation_config.add_target_language('hi')
print('Ready to translate from',translation_config.speech_recognition_language)
```

5. You will use the **SpeechTranslationConfig** to translate speech into text, but you will also use a **SpeechConfig** to synthesize translations into speech. Add the following code under the comment **Configure speech**:

```
Code
Copy

# Configure speech
speech_config = speech_sdk.SpeechConfig(speech_key, speech_region)
print('Ready to use speech service in:', speech_config.region)
```

6. Save your changes (*CTRL+S*), but leave the code editor open.

Run the app

So far, the app doesn't do anything other than connect to your Azure AI Speech resource, but it's useful to run it and check that it works before adding speech functionality.

1. In the command line, enter the following command to run the translator app:

```
Code
Copy

python translator.py
```

The code should display the region of the speech service resource the application will use, a message that it is ready to translate from en-US and prompt you for a target language. A successful run indicates that the app has connected to your Azure AI Speech service. Press ENTER to end the program.

Implement speech translation

Now that you have a **SpeechTranslationConfig** for the Azure AI Speech service, you can use the Azure AI Speech translation API to recognize and translate speech.

1. In the code file, note that the code uses the **Translate** function to translate spoken input. Then in the **Translate** function, under the comment **Translate speech**, add the following code to create a **TranslationRecognizer** client that can be used to recognize and translate speech from a file.

```
# Translate speech
current_dir = os.getcwd()
audioFile = current_dir + '/station.wav'
audio_config_in = speech_sdk.AudioConfig(filename=audioFile)
translator = speech_sdk.translation.TranslationRecognizer(translation_config, audio_config =
audio_config_in)
print("Getting speech from file...")
result = translator.recognize_once_async().get()
print('Translating "{}"'.format(result.text))
translation = result.translations[targetLanguage]
print(translation)
```

2. Save your changes (*CTRL+S*), and re-run the program:

```
python translator.py
```

3. When prompted, enter a valid language code (*fr*, *es*, or *hi*). The program should transcribe your input file and translate it to the language you specified (French, Spanish, or Hindi). Repeat this process, trying each language supported by the application.

> **NOTE**: The translation to Hindi may not always be displayed correctly in the Console window due to character encoding issues.

4. When you're finished, press ENTER to end the program.

> **NOTE**: The code in your application translates the input to all three languages in a single call. Only the translation for the specific language is displayed, but you could retrieve any of the translations by specifying the target language code in the **translations** collection of the result.

Synthesize the translation to speech

So far, your application translates spoken input to text; which might be sufficient if you need to ask someone for help while traveling. However, it would be better to have the translation spoken aloud in a suitable voice.

> **Note**: Due to the hardware limitations of the cloud shell, we'll direct the synthesized speech output to a file.

1. In the **Translate** function, find the comment **Synthesize translation**, and add the following code to use a **SpeechSynthesizer** client to synthesize the translation as speech and save it as a .wav file:

```
# Synthesize translation
output_file = "output.wav"
voices = {
        "fr": "fr-FR-HenriNeural",
        "es": "es-ES-ElviraNeural",
        "hi": "hi-IN-MadhurNeural"
}
speech_config.speech_synthesis_voice_name = voices.get(targetLanguage)
audio_config_out = speech_sdk.audio.AudioConfig(filename=output_file)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config_out)
speak = speech_synthesizer.speak_text_async(translation).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
else:
    print("Spoken output saved in " + output_file)
```

2. Save your changes (*CTRL+S*), and re-run the program:

Code Copy

```
python translator.py
```

3. Review the output from the application, which should indicate that the spoken output translation was saved in a file. When you're finished, press **ENTER** to end the program.

4. If you have a media player capable of playing .wav audio files, download the file that was generated by entering the following command:

Code Copy

```
download ./output.wav
```

The download command creates a popup link at the bottom right of your browser, which you can select to download and open the file.

> **NOTE** *In this example, you've used a **SpeechTranslationConfig** to translate speech to text, and then used a **SpeechConfig** to synthesize the translation as speech. You can in fact use the **SpeechTranslationConfig** to synthesize the translation directly, but this only works when translating to a single language, and results in an audio stream that is typically saved as a file.*

Clean up resources

If you're finished exploring the Azure AI Speech service, you can delete the resources you created in this exercise. Here's how:

1. Close the Azure cloud shell pane
2. In the Azure portal, browse to the Azure AI Speech resource you created in this lab.
3. On the resource page, select **Delete** and follow the instructions to delete the resource.

What if you have a mic and speaker?

In this exercise, the Azure Cloud Shell environment we used doesn't support audio hardware, so you used audio files for the speech input and output. Let's see how the code can be modified to use audio hardware if you have it available.

Using speech translation with a microphone

1. If you have a mic, you can use the following code to capture spoken input for speech translation:

Code Copy

```
# Translate speech
audio_config_in = speech_sdk.AudioConfig(use_default_microphone=True)
translator = speech_sdk.translation.TranslationRecognizer(translation_config, audio_config =
audio_config_in)
print("Speak now...")
result = translator.recognize_once_async().get()
print('Translating "{}"'.format(result.text))
translation = result.translations[targetLanguage]
print(translation)
```

> **Note**: The system default microphone is the default audio input, so you could also just omit the AudioConfig altogether!

Using speech synthesis with a speaker

1. If you have a speaker, you can use the following code to synthesize speech.

Code Copy

```
# Synthesize translation
voices = {
        "fr": "fr-FR-HenriNeural",
        "es": "es-ES-ElviraNeural",
        "hi": "hi-IN-MadhurNeural"
}
speech_config.speech_synthesis_voice_name = voices.get(targetLanguage)
audio_config_out = speech_sdk.audio.AudioOutputConfig(use_default_speaker=True)
speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config, audio_config_out)
speak = speech_synthesizer.speak_text_async(translation).get()
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
```

> **Note**: The system default speaker is the default audio output, so you could also just omit the AudioConfig altogether!

More information

For more information about using the Azure AI Speech translation API, see the Speech translation documentation.

Develop an audio-enabled chat app

In this exercise, you use the *Phi-4-multimodal-instruct* generative AI model to generate responses to prompts that include audio files. You'll develop an app that provides AI assistance for a produce supplier company by using Azure AI Foundry and the Python OpenAI SDK to summarize voice messages left by customers.

While this exercise is based on Python, you can develop similar applications using multiple language-specific SDKs; including:

Create an Azure AI Foundry project

Create a client application

Summary

Clean up

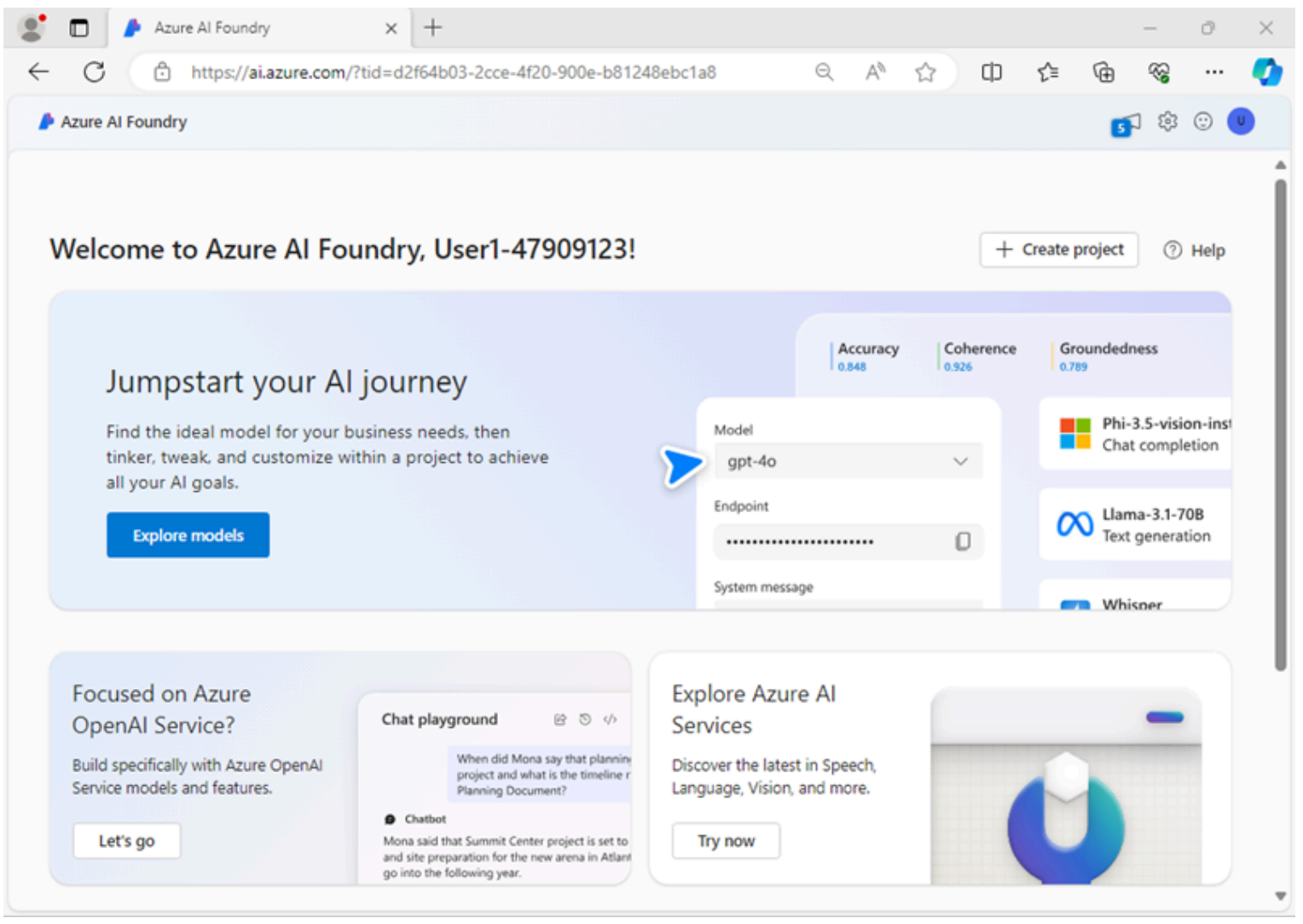
- Azure AI Projects for Python
- OpenAI library for Python
- Azure AI Projects for Microsoft .NET
- Azure OpenAI client library for Microsoft .NET
- Azure AI Projects for JavaScript
- Azure OpenAI library for TypeScript

This exercise takes approximately **30** minutes.

Create an Azure AI Foundry project

Let's start by deploying a model in an Azure AI Foundry project.

1. In a web browser, open the Azure AI Foundry portal at `https://ai.azure.com` and sign in using your Azure credentials. Close any tips or quick start panes that are opened the first time you sign in, and if necessary use the **Azure AI Foundry** logo at the top left to navigate to the home page, which looks similar to the following image:



2. In the home page, in the **Explore models and capabilities** section, search for the `Phi-4-multimodal-instruct` model; which we'll use in our project.
3. In the search results, select the **Phi-4-multimodal-instruct** model to see its details, and then at the top of the page for the model, select **Use this model**.
4. When prompted to create a project, enter a valid name for your project and expand **Advanced options**.
5. Select **Customize** and specify the following settings for your hub:
    - **Azure AI Foundry resource**: *A valid name for your Azure AI Foundry resource*

- **Subscription**: *Your Azure subscription*
- **Resource group**: *Create or select a resource group*
- **Region**: *Select any **AI Foundry recommended****

> * Some Azure AI resources are constrained by regional model quotas. In the event of a quota limit being exceeded later in the exercise, there's a possibility you may need to create another resource in a different region. You can check the latest regional availability for specific models in the Azure AI Foundry documentation

6. Select **Create** and wait for your project to be created.

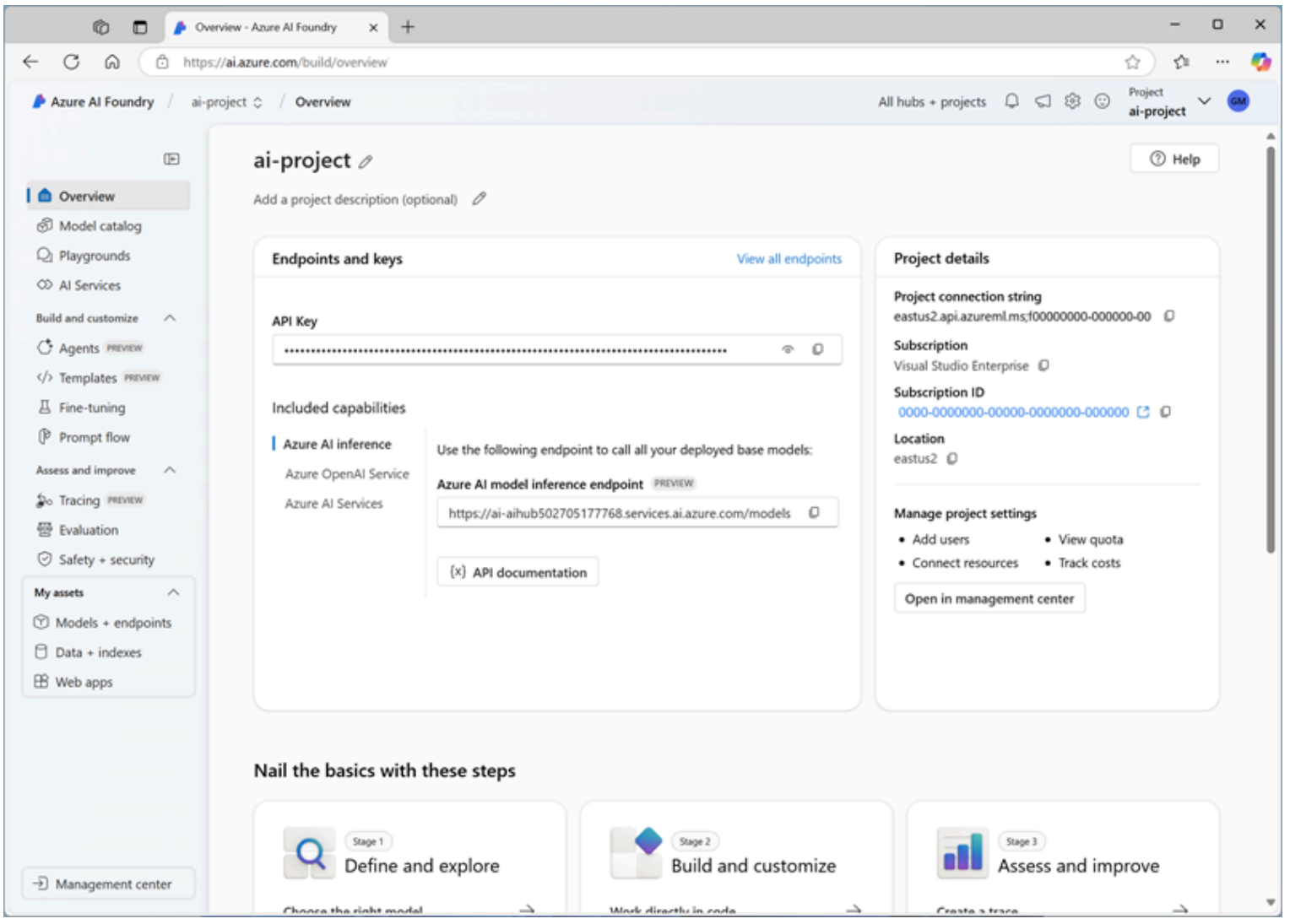
   It may take a few moments for the operation to complete.

7. Select **Agree and Proceed** to agree to the model terms, then select **Deploy** to complete the Phi model deployment.

8. When your project is created, the model details will be opened automatically. Note the name of your model deployment; which should be **Phi-4-multimodal-instruct**

9. In the navigation pane on the left, select **Overview** to see the main page for your project; which looks like this:

> **Note**: If an *Insufficient permissions** error is displayed, use the **Fix me** button to resolve it.



Create a client application

Now that you deployed a model, you can use the Azure AI Foundry and Azure AI Model Inference SDKs to develop an application that chats with it.

> **Tip**: You can choose to develop your solution using Python or Microsoft C#. Follow the instructions in the appropriate section for your chosen language.

Prepare the application configuration

1. In the Azure AI Foundry portal, view the **Overview** page for your project.

2. In the **Project details** area, note the **Azure AI Foundry project endpoint**. You'll use this endpoint to connect to your project in a client application.

3. Open a new browser tab (keeping the Azure AI Foundry portal open in the existing tab). Then in the new tab, browse to the Azure portal at `https://portal.azure.com`; signing in with your Azure credentials if prompted.

   Close any welcome notifications to see the Azure portal home page.

4. Use the **[>_]** button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a ***PowerShell*** environment with no storage in your subscription.

   The cloud shell provides a command-line interface in a pane at the bottom of the Azure portal. You can resize or maximize this pane to make it easier to work in.

   > **Note**: If you have previously created a cloud shell that uses a *Bash* environment, switch it to ***PowerShell***.

5. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

   Ensure you've switched to the classic version of the cloud shell before continuing.

6. In the cloud shell pane, enter the following commands to clone the GitHub repo containing the code files for this exercise (type the command, or copy it to the clipboard and then right-click in the command line and paste as plain text):

   Code Copy

   ```
   rm -r mslearn-ai-audio -f
   git clone https://github.com/MicrosoftLearning/mslearn-ai-language
   ```

   > **Tip**: As you paste commands into the cloudshell, the ouput may take up a large amount of the screen buffer. You can clear the screen by entering the `cls` command to make it easier to focus on each task.

7. After the repo has been cloned, navigate to the folder containing the application code files:

   Code Copy

   ```
   cd mslearn-ai-language/Labfiles/09-audio-chat/Python
   ```

8. In the cloud shell command line pane, enter the following command to install the libraries you'll use:

   Code Copy

   ```
   python -m venv labenv
   ./labenv/bin/Activate.ps1
   pip install -r requirements.txt azure-identity azure-ai-projects openai
   ```

9. Enter the following command to edit the configuration file that has been provided:

   Code Copy

   ```
   code .env
   ```

   The file should open in a code editor.

10. In the code file, replace the **your_project_endpoint** placeholder with the endpoint for your project (copied from the project **Overview** page in the Azure AI Foundry portal), and the **your_model_deployment** placeholder with the name you assigned to your Phi-4-multimodal-instruct model deployment.

11. After you replace the placeholders, in the code editor, use the **CTRL+S** command or **Right-click > Save** to save your changes and then use the **CTRL+Q** command or **Right-click > Quit** to close the code editor while keeping the cloud shell command line open.

Write code to connect to your project and get a chat client for your model

> **Tip**: As you add code, be sure to maintain the correct indentation.

1. Enter the following command to edit the code file:

```
code audio-chat.py
```

2. In the code file, note the existing statements that have been added at the top of the file to import the necessary SDK namespaces. Then, Find the comment **Add references**, add the following code to reference the namespaces in the libraries you installed previously:

```
# Add references
from azure.identity import DefaultAzureCredential
from azure.ai.projects import AIProjectClient
```

3. In the **main** function, under the comment **Get configuration settings**, note that the code loads the project connection string and model deployment name values you defined in the configuration file.

4. Find the comment **Initialize the project client** and add the following code to connect to your Azure AI Foundry project:

> **Tip**: Be careful to maintain the correct indentation level for your code.

```
# Initialize the project client
project_client = AIProjectClient(
   credential=DefaultAzureCredential(
       exclude_environment_credential=True,
       exclude_managed_identity_credential=True
   ),
   endpoint=project_endpoint,
)
```

5. Find the comment **Get a chat client**, add the following code to create a client object for chatting with your model:

```
# Get a chat client
openai_client = project_client.get_openai_client(api_version="2024-10-21")
```

Write code to submit an audio-based prompt

Before submitting the prompt, we need to encode the audio file for the request. Then we can attach the audio data to the user's message with a prompt for the LLM. Note that the code includes a loop to allow the user to input a prompt until they enter "quit".

1. Under the comment **Encode the audio file**, enter the following code to prepare the following audio file:

Code Copy

```
# Encode the audio file
file_path = "https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/Labfiles/09-audio-chat/data/avocados.mp3"
response = requests.get(file_path)
response.raise_for_status()
audio_data = base64.b64encode(response.content).decode('utf-8')
```

2. Under the comment **Get a response to audio input**, add the following code to submit a prompt:

Code Copy

```
# Get a response to audio input
response = openai_client.chat.completions.create(
   model=model_deployment,
   messages=[
       {"role": "system", "content": system_message},
       { "role": "user",
           "content": [
           {
               "type": "text",
               "text": prompt
           },
           {
               "type": "input_audio",
               "input_audio": {
                   "data": audio_data,
                   "format": "mp3"
               }
           }
       ] }
   ]
)
print(response.choices[0].message.content)
```

3. Use the **CTRL+S** command to save your changes to the code file. You can also close the code editor (**CTRL+Q**) if you like.

Sign into Azure and run the app

1. In the cloud shell command-line pane, enter the following command to sign into Azure.

Code Copy

```
az login
```

You must sign into Azure - even though the cloud shell session is already authenticated.

> **Note**: In most scenarios, just using *az login* will be sufficient. However, if you have subscriptions in multiple tenants, you may need to specify the tenant by using the *--tenant* parameter. See Sign into Azure interactively using the Azure CLI for details.

2. When prompted, follow the instructions to open the sign-in page in a new tab and enter the authentication code provided and your Azure credentials. Then complete the sign in process in the command line, selecting the subscription containing your Azure AI Foundry hub if prompted.
3. In the cloud shell command-line pane, enter the following command to run the app:

```
python audio-chat.py
```

4. When prompted, enter the prompt

```
Can you summarize this customer's voice message?
```

5. Review the response.

Use a different audio file

1. In the code editor for your app code, find the code you added previously under the comment **Encode the audio file**. Then modify the file path url as follows to use a different audio file for the request (leaving the existing code after the file path):

```
# Encode the audio file
file_path = "https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/Labfiles/09-audio-chat/data/fresas.mp3"
```

The new file sounds like this:

2. Use the **CTRL+S** command to save your changes to the code file. You can also close the code editor (**CTRL+Q**) if you like.
3. In the cloud shell command line pane beneath the code editor, enter the following command to run the app:

```
python audio-chat.py
```

4. When prompted, enter the following prompt:

```
Can you summarize this customer's voice message? Is it time-sensitive?
```

5. Review the response. Then enter `quit` to exit the program.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

6. You can continue to run the app, choosing different prompt types and trying different prompts. When you're finished, enter `quit` to exit the program.

   If you have time, you can modify the code to use a different system prompt and your own internet-accessible audio files.

> **Note**: In this simple app, we haven't implemented logic to retain conversation history; so the model will treat each prompt as a new request with no context of the previous prompt.

Summary

In this exercise, you used Azure AI Foundry and the Azure AI Inference SDK to create a client application uses a multimodal model to generate responses to audio.

Clean up

If you've finished exploring Azure AI Foundry, you should delete the resources you have created in this exercise to avoid incurring unnecessary Azure costs.

1. Return to the browser tab containing the Azure portal (or re-open the Azure portal at `https://portal.azure.com` in a new browser tab) and view the contents of the resource group where you deployed the resources used in this exercise.
2. On the toolbar, select **Delete resource group**.
3. Enter the resource group name and confirm that you want to delete it.

Launch the Azure Cloud Shell and download the files

Add code to complete the web app

Update and run the deployment script

View and test the app

Clean up resources

Develop an Azure AI Voice Live voice agent

In this exercise, you complete a Flask-based Python web app based that enables real-time voice interactions with an agent. You add the code to initialize the session, and handle session events. You use a deployment script that: deploys the AI model; creates an image of the app in Azure Container Registry (ACR) using ACR tasks; and then creates an Azure App Service instance that pulls the the image. To test the app you will need an audio device with microphone and speaker capabilities.

While this exercise is based on Python, you can develop similar applications other language-specific SDKs; including:

- Azure VoiceLive client library for .NET

Tasks performed in this exercise:

- Download the base files for the app
- Add code to complete the web app
- Review the overall code base
- Update and run the deployment script
- View and test the application

This exercise takes approximately **30** minutes to complete.

Launch the Azure Cloud Shell and download the files

In this section of the exercise you download the a zipped file containing the base files for the app.

1. In your browser navigate to the Azure portal https://portal.azure.com; signing in with your Azure credentials if prompted.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new cloud shell in the Azure portal, selecting a ***Bash*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *PowerShell* environment, switch it to ***Bash***.

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

4. Run the following command in the **Bash** shell to download and unzip the exercise files. The second command will also change to the directory for the exercise files.

Code Copy

```
wget https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/downloads/python/voice-live-web.zip
```

Code Copy

```
unzip voice-live-web.zip && cd voice-live-web
```

Add code to complete the web app

Now that the exercise files are downloaded, the the next step is to add code to complete the application. The following steps are performed in the cloud shell.

> **Tip:** Resize the cloud shell to display more information, and code, by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

Run the following command to change into the *src* directory before you continue with the exercise.

Code Copy

```
cd src
```

Add code to implement the voice live assistant

In this section you add code to implement the voice live assistant. The **__init__** method initializes the voice assistant by storing the Azure VoiceLive connection parameters (endpoint, credentials, model, voice, and system instructions) and setting up runtime state variables to manage the connection lifecycle and handle user interruptions during conversations. The **start** method imports the necessary Azure VoiceLive SDK components that will be used to establish the WebSocket connection and configure the real-time voice session.

1. Run the following command to open the *flask_app.py* file for editing.

   Code Copy

   ```
   code flask_app.py
   ```

2. Search for the **# BEGIN VOICE LIVE ASSISTANT IMPLEMENTATION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

   Code Copy

```
def __init__(
    self,
    endpoint: str,
    credential,
    model: str,
    voice: str,
    instructions: str,
    state_callback=None,
):
    # Store Azure Voice Live connection and configuration parameters
    self.endpoint = endpoint
    self.credential = credential
    self.model = model
    self.voice = voice
    self.instructions = instructions

    # Initialize runtime state - connection established in start()
    self.connection = None
    self._response_cancelled = False  # Used to handle user interruptions
    self._stopping = False  # Signals graceful shutdown
    self.state_callback = state_callback or (lambda *_: None)

async def start(self):
    # Import Voice Live SDK components needed for establishing connection and configuring
session
    from azure.ai.voicelive.aio import connect  # type: ignore
    from azure.ai.voicelive.models import (
        RequestSession,
        ServerVad,
        AzureStandardVoice,
        Modality,
        InputAudioFormat,
        OutputAudioFormat,
    )  # type: ignore
```

3. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to implement the voice live assistant

In this section you add code to configure the voice live session. This specifies the modalities (audio-only is not supported by the API), the system instructions that define the assistant's behavior, the Azure TTS voice for responses, the audio format for both input and output streams, and Server-side Voice Activity Detection (VAD) which specifies how the model detects when users start and stop speaking.

1. Search for the **# BEGIN CONFIGURE VOICE LIVE SESSION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

Code Copy

```
# Configure VoiceLive session with audio/text modalities and voice activity detection
session_config = RequestSession(
    modalities=[Modality.TEXT, Modality.AUDIO],
    instructions=self.instructions,
    voice=voice_cfg,
    input_audio_format=InputAudioFormat.PCM16,
    output_audio_format=OutputAudioFormat.PCM16,
    turn_detection=ServerVad(threshold=0.5, prefix_padding_ms=300, silence_duration_ms=500),
)
await conn.session.update(session=session_config)
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to handle session events

In this section you add code to add event handlers for the voice live session. The event handlers respond to key VoiceLive session events during the conversation lifecycle: **_handle_session_updated** signals when the session is ready for user input, **_handle_speech_started** detects when the user begins speaking and implements interruption logic by stopping any ongoing assistant audio playback and canceling in-progress responses to allow natural conversation flow, and **_handle_speech_stopped** handles when the user has finished speaking and the assistant begins processing the input.

1. Search for the **# BEGIN HANDLE SESSION EVENTS - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment, be sure to check the indentation.

Code Copy

```
async def _handle_event(self, event, conn, verbose=False):
    """Handle Voice Live events with clear separation by event type."""
    # Import event types for processing different Voice Live server events
    from azure.ai.voicelive.models import ServerEventType

    event_type = event.type
    if verbose:
        _broadcast({"type": "log", "level": "debug", "event_type": str(event_type)})

    # Route Voice Live server events to appropriate handlers
    if event_type == ServerEventType.SESSION_UPDATED:
        await self._handle_session_updated()
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STARTED:
        await self._handle_speech_started(conn)
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STOPPED:
        await self._handle_speech_stopped()
    elif event_type == ServerEventType.RESPONSE_AUDIO_DELTA:
        await self._handle_audio_delta(event)
    elif event_type == ServerEventType.RESPONSE_AUDIO_DONE:
        await self._handle_audio_done()
    elif event_type == ServerEventType.RESPONSE_DONE:
        # Reset cancellation flag but don't change state - _handle_audio_done already did
        self._response_cancelled = False
    elif event_type == ServerEventType.ERROR:
        await self._handle_error(event)

async def _handle_session_updated(self):
    """Session is ready for conversation."""
    self.state_callback("ready", "Session ready. You can start speaking now.")

async def _handle_speech_started(self, conn):
    """User started speaking - handle interruption if needed."""
    self.state_callback("listening", "Listening… speak now")

    try:
        # Stop any ongoing audio playback on the client side
        _broadcast({"type": "control", "action": "stop_playback"})

        # If assistant is currently speaking or processing, cancel the response to allow
interruption
        current_state = assistant_state.get("state")
        if current_state in {"assistant_speaking", "processing"}:
            self._response_cancelled = True
            await conn.response.cancel()
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"Interrupted assistant during {current_state}"})
        else:
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"User speaking during {current_state} - no cancellation
needed"})
    except Exception as e:
        _broadcast({"type": "log", "level": "debug",
                    "msg": f"Exception in speech handler: {e}"})

async def _handle_speech_stopped(self):
    """User stopped speaking - processing input."""
    self.state_callback("processing", "Processing your input…")
```

```
async def _handle_audio_delta(self, event):
    """Stream assistant audio to clients."""
    if self._response_cancelled:
        return  # Skip cancelled responses

    # Update state when assistant starts speaking
    if assistant_state.get("state") != "assistant_speaking":
        self.state_callback("assistant_speaking", "Assistant speaking…")

    # Extract and broadcast Voice Live audio delta as base64 to WebSocket clients
    audio_data = getattr(event, "delta", None)
    if audio_data:
        audio_b64 = base64.b64encode(audio_data).decode("utf-8")
        _broadcast({"type": "audio", "audio": audio_b64})

async def _handle_audio_done(self):
    """Assistant finished speaking."""
    self._response_cancelled = False
    self.state_callback("ready", "Assistant finished. You can speak again.")

async def _handle_error(self, event):
    """Handle Voice Live errors."""
    error = getattr(event, "error", None)
    message = getattr(error, "message", "Unknown error") if error else "Unknown error"
    self.state_callback("error", f"Error: {message}")

def request_stop(self):
    self._stopping = True
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Review the code in the app

So far, you've added code to the app to implement the agent and handle agent events. Take a few minutes to review the full code and comments to get a better understanding of how the app is handling client state and operations.

1. When you're finished enter **ctrl+q** to exit out of the editor.

Update and run the deployment script

In this section you make a small change to the **azdeploy.sh** deployment script and then run the deployment.

Update the deployment script

There are only two values you should change at the top of the **azdeploy.sh** deployment script.

- The **rg** value specifies the resource group to contain the deployment. You can accept the default value, or enter your own value if you need to deploy to a specific resource group.
- The **location** value sets the region for the deployment. The *gpt-4o* model used in the exercise can be deployed to other regions, but there can be limits in any particular region. If the deployment fails in your chosen region, try **eastus2** or **swedencentral**.

Code Copy

```
rg="rg-voicelive" # Replace with your resource group
location="eastus2" # Or a location near you
```

1. Run the following commands in the Cloud Shell to begin editing the deployment script.

```
cd ~/voice-live-web
```

```
code azdeploy.sh
```

2. Update the values for **rg** and **location** to meet your needs and then enter **ctrl+s** to save your changes and **ctrl+q** to exit the editor.

Run the deployment script

The deployment script deploys the AI model and creates the necessary resources in Azure to run a containerized app in App Service.

1. Run the following command in the Cloud Shell to begin deploying the Azure resources and the application.

```
bash azdeploy.sh
```

2. Select **option 1** for the initial deployment.

   The deployment should complete in 5-10 minutes. During the deployment you might be prompted for the following information/actions:

   - If you are prompted to authenticate to Azure follow the directions presented to you.
   - If you are prompted to select a subscription use the arrow keys to highlight your subscription and press **Enter**.
   - You will likely see some warnings during deployment and these can be ignored.
   - If the deployment fails during the AI model deployment change the region in the deployment script and try again.
   - Regions in Azure can get busy at times and interrupt the timing of the deployments. If the deployment fails after the model deployment re-run the deployment script.

View and test the app

When the deployment completes a "Deployment complete!" message will be in the shell along with a link to the web app. You can select that link, or navigate to the App Service resource and launch the app from there. It can take a few minutes for the application to load.

1. Select the **Start session** button to connect to the model.
2. You will be prompted to give the application access to you audio devices.
3. Begin talking to the model when the app prompts you to start speaking.

Troubleshooting:

- If the app reports missing environment variables, restart the application in App Service.
- If you see excessive *audio chunk* messages in the log shown in the application select **Stop session** and then start the session again.
- If the app fails to function at all, double-check you added all of the code and for proper indentation. If you need to make any changes re-run the deployment and select **option 2** to only update the image.

Clean up resources

Run the following command in the Cloud Shell to remove all of the resources deployed for this exercise. You will be prompted to confirm the resource deletion.

Code Copy

```
azd down --purge
```