Create a Language Understanding Client Application

The Language Understanding service enables you to define an app that encapsulates a language model that applications can use to interpret natural language input from users, predict the users *intent* (what they want to achieve), and identify any *entities* to which the intent should be applied. You can create client applications that consume Language Understanding apps directly through REST interfaces, or by using language -specific software development kits (SDKs).

Clone the repository for this course

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

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLearning/AI-102-AIEngineer` repository to a local folder (it doesn't matter which folder).
3. When the repository has been cloned, open the folder in Visual Studio Code.

4. Wait while additional files are installed to support the C# code projects in the repo.

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

Create Language Understanding resources

If you already have Language Understanding authoring and prediction resources in your Azure subscription, you can use them in this exercise. Otherwise, follow these instructions to create them.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. Select the **+Create a resource** button, search for *language understanding*, and create a **Language Understanding** resource with the following settings:
   - **Create option**: Both
   - **Subscription**: *Your Azure subscription*
   - **Resource group**: *Choose or create a resource group (if you are using a restricted subscription, you may not have permission to create a new resource group - use the one provided)*
   - **Name**: *Enter a unique name*
   - **Authoring location**: *Select your preferred location*
   - **Authoring pricing tier**: F0
   - **Prediction location**: *Choose the same location as your authoring location*
   - **Prediction pricing tier**: F0 (*If F0 is not available, choose S0*)
3. Wait for the resources to be created, and note that two Language Understanding resources are provisioned; one for authoring, and another for prediction. You can view both of these by navigating to the resource group where you created them.

Import, train, and publish a Language Understanding app

If you already have a **Clock** app from a previous exercise, you can use it in this exercise. Otherwise, follow these instructions to create it.

1. In a new browser tab, open the Language Understanding portal at `https://www.luis.ai`.
2. Sign in using the Microsoft account associated with your Azure subscription. If this is the first time you have signed into the Language Understanding portal, you may need to grant the app some permissions to

access your account details. Then complete the *Welcome* steps by selecting your Azure subscription and the authoring resource you just created.
3. Open the **Conversation Apps** page, next to **+ New app**, view the drop-down list and select **Import As LU**. Browse to the **10-luis-client** subfolder in the project folder containing the lab files for this exercise, and select **Clock.lu**. Then specify a unique name for the clock app.
4. If a panel with tips for creating an effective Language Understanding app is displayed, close it.
5. At the top of the Language Understanding portal, select **Train** to train the app.
6. At the top right of the Language Understanding portal, select **Publish** and publish the app to the **Production slot**.
7. After publishing is complete, at the top of the Language Understanding portal, select **Manage**.
8. On the **Settings** page, note the **App ID**. Client applications need this to use your app.
9. On the **Azure Resources** page, under **Prediction resources**, if no prediction resource is listed, add the prediction resource in your Azure subscription.
10. Note the **Primary Key**, **Secondary Key**, and **Endpoint URL** for the prediction resource. Client applications need the endpoint and one of the keys to connect to the prediction resource and be authenticated.

Prepare to use the Language Understanding SDK

In this exercise, you'll complete a partially implemented client application that uses the clock Language Understanding app to predict intents from user input and respond appropriately.

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

1. In Visual Studio Code, in the **Explorer** pane, browse to the **10-luis-client** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **clock-client** folder and open an integrated terminal. Then install the Language Understanding SDK package by running the appropriate command for your language preference:

C#

Code | Copy

```
dotnet add package Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime --version 3.0.0
```

*In addition to the **Runtime** (prediction) package, there is an **Authoring** package that you can use to write code to create and manage Language Understanding models.*

Python

Code | Copy

```
pip install azure-cognitiveservices-language-luis==0.7.0
```

*The Python SDK package includes classes for both **prediction** and **authoring**.*

1. View the contents of the **clock-client** folder, and note that it contains a file for configuration settings:

   - **C#**: appsettings.json
   - **Python**: .env

   Open the configuration file and update the configuration values it contains to include the **APP ID** for your Language Understanding app, and the **Endpoint URL** and one of the **Keys** for its prediction resource (from the **Manage** page for your app in the Language Understanding portal).

2. Note that the **clock-client** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: clock-client.py

Open the code file and at the top, 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 Language Understanding prediction SDK:

C#

Code Copy

```
// Import namespaces
using Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime;
using Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime.Models;
```

Python

Code Copy

```
# Import namespaces
from azure.cognitiveservices.language.luis.runtime import LUISRuntimeClient
from msrest.authentication import CognitiveServicesCredentials
```

Get a prediction from the Language Understanding app

Now you're ready to implement code that uses the SDK to get a prediction from your Language Understanding app.

1. In the **Main** function, note that code to load the App ID, prediction endpoint, and key from the configuration file has already been provided. Then find the comment **Create a client for the LU app** and add the following code to create a prediction client for your Language Understanding app:

C#

Code Copy

```
// Create a client for the LU app
var credentials = new
Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime.ApiKeyServiceClientCredentials(predictionKe

var luClient = new LUISRuntimeClient(credentials) { Endpoint = predictionEndpoint };
```

Python

Code Copy

```
# Create a client for the LU app
credentials = CognitiveServicesCredentials(lu_prediction_key)
lu_client = LUISRuntimeClient(lu_prediction_endpoint, credentials)
```

1. 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 LU app to get intent and entities** and add the following code:

C#

Code Copy

```
// Call the LU app to get intent and entities
var slot = "Production";
var request = new PredictionRequest { Query = userText };
PredictionResponse predictionResponse = await luClient.Prediction.GetSlotPredictionAsync(luAppId,
slot, request);
Console.WriteLine(JsonConvert.SerializeObject(predictionResponse, Formatting.Indented));
Console.WriteLine("--------------------\n");
Console.WriteLine(predictionResponse.Query);
var topIntent = predictionResponse.Prediction.TopIntent;
var entities = predictionResponse.Prediction.Entities;
```

Python

Code Copy

```
# Call the LU app to get intent and entities
request = { "query" : userText }
slot = 'Production'
prediction_response = lu_client.prediction.get_slot_prediction(lu_app_id, slot, request)
top_intent = prediction_response.prediction.top_intent
entities = prediction_response.prediction.entities
print('Top Intent: {}'.format(top_intent))
print('Entities: {}'.format (entities))
print('-----------------\n{}'.format(prediction_response.query))
```

The call to the Language Understanding app returns a prediction, 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.

1. 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.

C#

Code Copy

```
// Apply the appropriate action
switch (topIntent)
{
    case "GetTime":
        var location = "local";
        // Check for entities
        if (entities.Count > 0)
        {
            // Check for a location entity
            if (entities.ContainsKey("Location"))
            {
                //Get the JSON for the entity
                var entityJson = JArray.Parse(entities["Location"].ToString());
                // ML entities are strings, get the first one
                location = entityJson[0].ToString();
            }
        }

        // Get the time for the specified location
        var getTimeTask = Task.Run(() => GetTime(location));
        string timeResponse = await getTimeTask;
        Console.WriteLine(timeResponse);
        break;

    case "GetDay":
        var date = DateTime.Today.ToShortDateString();
        // Check for entities
        if (entities.Count > 0)
        {
            // Check for a Date entity
            if (entities.ContainsKey("Date"))
            {
                //Get the JSON for the entity
                var entityJson = JArray.Parse(entities["Date"].ToString());
                // Regex entities are strings, get the first one
                date = entityJson[0].ToString();
            }
        }
        // Get the day for the specified date
        var getDayTask = Task.Run(() => GetDay(date));
        string dayResponse = await getDayTask;
        Console.WriteLine(dayResponse);
        break;

    case "GetDate":
        var day = DateTime.Today.DayOfWeek.ToString();
        // Check for entities
        if (entities.Count > 0)
        {
            // Check for a Weekday entity
            if (entities.ContainsKey("Weekday"))
            {
                //Get the JSON for the entity
                var entityJson = JArray.Parse(entities["Weekday"].ToString());
                // List entities are lists
                day = entityJson[0][0].ToString();
            }
        }
        // Get the date for the specified day
        var getDateTask = Task.Run(() => GetDate(day));
        string dateResponse = await getDateTask;
        Console.WriteLine(dateResponse);
        break;

    default:
        // Some other intent (for example, "None") was predicted
        Console.WriteLine("Try asking me for the time, the day, or the date.");
        break;
}
```

Python

Code

```
# Apply the appropriate action
if top_intent == 'GetTime':
    location = 'local'
    # Check for entities
    if len(entities) > 0:
        # Check for a location entity
        if 'Location' in entities:
            # ML entities are strings, get the first one
            location = entities['Location'][0]
    # 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
        if 'Date' in entities:
            # Regex entities are strings, get the first one
            date_string = entities['Date'][0]
    # 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
        if 'Weekday' in entities:
            # List entities are lists
            day = entities['Weekday'][0][0]
    # 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.')
```

1. Save your changes and return to the integrated terminal for the **clock-client** folder, and enter the following command to run the program:

C#

Code

```
dotnet run
```

Python

Code

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

1. 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 Understanding in which your application must:

1. Connect to a prediction endpoint.
2. Submit an utterance to get a prediction.
3. Implement logic to respond appropriately to the predicted intent and entities.

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

More information

To learn more about creating a Language Understanding client, see the developer documentation