Contents

1 AI-102: AI Engineer **7**
- 1.1 What are we doing? 7
- 1.2 How should I use these files relative to the released MOC files? 8
- 1.3 What about changes to the student handbook? 8
- 1.4 How do I contribute? 8
- 1.5 title: Online Hosted Instructions permalink: index.html layout: home 8

2 Content Directory **8**
- 2.1 Labs 8

3 The MIT License (MIT) **8**
- 3.1 THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. 9
- 3.2 lab: title: ’Lab Environment Setup’ module: ’Setup’ 9

4 Lab Environment Setup **9**
- 4.0.1 Base Operating System (Windows 10) 9
  - 4.0.1.1 Windows 10 9
  - 4.0.1.2 Edge 9
- 4.0.2 .NET Core SDK 9
- 4.0.3 C++ Redistributable 9
- 4.0.4 Node.JS 9
- 4.0.5 Python (and required packages) 9
- 4.0.6 Azure CLI 9
- 4.0.7 Git 9
- 4.0.8 Visual Studio Code (and extensions) 10
- 4.0.9 Bot Framework Emulator 10
- 4.0.10 Bot Framework Composer 10
- 4.1 Install from https://docs.microsoft.com/en-us/composer/install-composer. 10
- 4.2 lab: title: ’Enable Resource Providers’ module: ’Setup’ 10

5 Enable Resource Providers **10**
- 5.1 lab: title: ’Get Started with Cognitive Services’ module: ’Module 2 - Developing AI Apps with Cognitive Services’ 10

6 Get Started with Cognitive Services **10**
- 6.1 Clone the repository for this course 11
- 6.2 Provision a Cognitive Services resource 11
- 6.3 Use a REST Interface 11
- 6.4 Use an SDK 12
- 6.5 More information 13
- 6.6 For more information about Azure Cognitive Services, see the Cognitive Services documentation. 13
- 6.7 lab: title: ’Manage Cognitive Services Security’ module: ’Module 2 - Developing AI Apps with Cognitive Services’ 13

7 Manage Cognitive Services Security **13**
- 7.1 Clone the repository for this course 13
- 7.2 Provision a Cognitive Services resource 13
- 7.3 Manage authentication keys 14
- 7.4 Secure key access with Azure Key Vault 15
  - 7.4.1 Create a key vault and add a secret 15
  - 7.4.2 Create a service principal 15
  - 7.4.3 Use the service principal in an application 16
- 7.5 More information 17

7.6 For more information about securing cognitive services, see the Cognitive Services security documentation. 17
7.7 lab: title: 'Monitor Cognitive Services' module: 'Module 2 - Developing AI Apps with Cognitive Services' 17

8 Monitor Cognitive Services 17
8.1 Clone the repository for this course 17
8.2 Provision a Cognitive Services resource 17
8.3 Configure an alert 17
8.4 Visualize a metric 18
8.5 More information 19
8.6 One of the options for monitoring cognitive services is to use *diagnostic logging*. Once enabled, diagnostic logging captures rich information about your cognitive services resource usage, and can be a useful monitoring and debugging tool. It can take over an hour after setting up diagnostic logging to generate any information, which is why we haven't explored it in this exercise; but you can learn more about it in the Cognitive Services documentation. 19
8.7 lab: title: 'Use a Cognitive Services Container' module: 'Module 2 - Developing AI Apps with Cognitive Services' 19

9 Use a Cognitive Services Container 19
9.1 Clone the repository for this course 19
9.2 Provision a Cognitive Services resource 19
9.3 Deploy and run a Text Analytics container 20
9.4 Use the container 21
9.5 Clean Up 21
9.6 More information 21
9.7 For more information about containerizing cognitive services, see the Cognitive Services containers documentation. 21
9.8 lab: title: 'Analyze Text' module: 'Module 3 - Getting Started with Natural Language Processing' 21

10 Analyze Text 21
10.1 Clone the repository for this course 21
10.2 Provision a Cognitive Services resource 21
10.3 Prepare to use the Text Analytics SDK 22
10.4 Detect language 23
10.5 Evaluate sentiment 23
10.6 Identify key phrases 24
10.7 Extract entities 25
10.8 Extract linked entities 25
10.9 More information 26
10.10 For more information about using the **Text Analytics** service, see the Text Analytics documentation. 26
10.11 lab: title: 'Translate Text' module: 'Module 3 - Getting Started with Natural Language Processing' 26

11 Translate Text 26
11.1 Clone the repository for this course 26
11.2 Provision a Cognitive Services resource 26
11.3 Prepare to use the Translator service 27
11.4 Detect language 27
11.5 Translate text 28
11.6 More information 30
11.7 For more information about using the **Translator** service, see the Translator documentation. . . 30
11.8 lab: title: 'Recognize and Synthesize Speech' module: 'Module 4 - Building Speech-Enabled Applications' 30

12 Recognize and Synthesize Speech 30
12.1 Clone the repository for this course 30
12.2 Provision a Cognitive Services resource 30
12.3 Prepare to use the Speech service 31
12.4 Recognize speech 32
12.4.1 If you have a working microphone 32

12.4.2 Alternatively, use audio input from a file 32
12.4.3 Add code to process the transcribed command 33
12.5 Synthesize speech 34
12.6 Use a different voice 34
12.7 Use Speech Synthesis Markup Language 35
12.8 More information 36
12.9 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. 36
12.10lab: title: 'Translate Speech' module: 'Module 4 - Building Speech-Enabled Applications' 36

13 Translate Speech **36**
13.1 Clone the repository for this course 36
13.2 Provision a Cognitive Services resource 36
13.3 Prepare to use the Speech Translation service 37
13.4 Implement speech translation 38
13.4.1 If you have a working microphone 38
13.4.2 Alternatively, use audio input from a file 39
13.4.3 Run the program 40
13.5 Synthesize the translation to speech 40
13.6 More information 41
13.7 For more information about using the **Speech translation** API, see the Speech translation documentation. 41
13.8 lab: title: 'Create a Language Understanding App' module: 'Module 5 - Creating Language Understanding Solutions' 41

14 Create a Language Understanding App **41**
14.1 Clone the repository for this course 41
14.2 Create Language Understanding resources 42
14.3 Create a Language Understanding app 42
14.4 Create intents 43
14.5 Train and test the app 43
14.6 Add entities 43
14.6.1 Add a *machine learned* entity 44
14.6.2 Add a *list* entity 44
14.6.3 Add a *Regex* entity 44
14.6.4 Retrain the app 45
14.7 Perform batch testing 45
14.8 Publish the app 46
14.9 Apply *active learning* 47
14.10Export the app 47
14.11More information 47
14.12For more information about using the **Language Understanding** service, see the Language Understanding documentation. 47
14.13lab: title: 'Create a Language Understanding Client Application' module: 'Module 5 - Creating Language Understanding Solutions' 47

15 Create a Language Understanding Client Application **47**
15.1 Clone the repository for this course 48
15.2 Create Language Understanding resources 48
15.3 Import, train, and publish a Language Understanding app 48
15.4 Prepare to use the Language Understanding SDK 49
15.5 Get a prediction from the Language Understanding app 49
15.6 More information 53
15.7 To learn more about creating a Language Understanding client, see the developer documentation 53
15.8 lab: title: 'Use the Speech and Language Understanding Services' module: 'Module 5 - Creating Language Understanding Solutions' 53

16 Use the Speech and Language Understanding Services **53**
16.1 Clone the repository for this course 53
16.2 Create Language Understanding resources 53
16.3 Prepare a Language Understanding app 53

16.4 Train and publish the app with *Speech Priming* . . . 54
16.5 Configure a client application for Language Understanding . . . 54
16.6 Install SDK Packages . . . 54
16.7 Create an *IntentRecognizer* . . . 55
16.7.1 **If you have a working microphone:** . . . 55
16.7.2 **If you need to use an audio file:** . . . 55
16.8 Get a predicted intent from spoken input . . . 56
16.9 Run the client application . . . 60
16.10More information . . . 60
16.11To learn more about Speech and Language Understanding integration, see the Speech documentation. . . . 60
16.12lab: title: 'Create a QnA Solution' module: 'Module 6 - Building a QnA Solution' . . . 60

17 Create a QnA Solution **60**
17.1 Create a QnA Maker resource . . . 60
17.2 Create a knowledge base . . . 61
17.3 Modify the knowledge base . . . 61
17.4 Train and test the knowledge base . . . 62
17.5 Publish the knowledge base . . . 62
17.6 Create a bot for the knowledge base . . . 62
17.7 More information . . . 63
17.8 To learn more about QnA Maker, see the QnA Maker documentation. . . . 63
17.9 lab: title: 'Create a Bot with the Bot Framework SDK' module: 'Module 7 - Conversational AI and the Azure Bot Service' . . . 63

18 Create a Bot with the Bot Framework SDK **63**
18.1 Before you start . . . 63
18.1.1 Update the Bot Framework Emulator . . . 63
18.1.2 Clone the repository for this course . . . 63
18.2 Create a bot . . . 63
18.3 Test the bot in the Bot Framework Emulator . . . 64
18.4 Modify the bot code . . . 64
18.5 If time permits: Deploy the bot to Azure . . . 66
18.5.1 Create or select a resource group . . . 66
18.5.2 Create an Azure application registration . . . 66
18.5.3 Create Azure resources . . . 66
18.5.4 Prepare the bot code for deployment . . . 67
18.5.5 Create a zip archive for deployment . . . 67
18.5.6 Deploy and test the bot . . . 67
18.6 Use the Web Chat channel in a web page . . . 67
18.7 More information . . . 68
18.8 To learn more about the Bot Framework, view the Bot Framework documentation. . . . 68
18.9 lab: title: 'Create a Bot with Bot Framework Composer' module: 'Module 7 - Conversational AI and the Azure Bot Service' . . . 68

19 Create a Bot with Bot Framework Composer **68**
19.1 Prepare to develop a bot . . . 68
19.1.1 Get an OpenWeather API key . . . 68
19.1.2 Update Bot Framework Composer . . . 68
19.2 Create a bot . . . 68
19.2.1 Create a bot and customize the "welcome" dialog flow . . . 68
19.2.2 Test the bot . . . 70
19.3 Add a dialog to get the weather . . . 70
19.3.1 Add a dialog . . . 70
19.3.2 Add a trigger for the dialog . . . 72
19.4 Handle interruptions . . . 73
19.5 Enhance the user experience . . . 74
19.5.1 Add a button . . . 74
19.5.2 Add a card . . . 74
19.5.3 Test the new user interface . . . 75
19.6 More information . . . 75

19.7 To learn more about Bot Framework Composer, view the Bot Framework Composer documentation. 75
19.8 lab: title: 'Analyze Images with Computer Vision' module: 'Module 8 - Getting Started with Computer Vision' . . . 75

20 Analyze Images with Computer Vision 75
20.1 Clone the repository for this course . . . 75
20.2 Provision a Cognitive Services resource . . . 75
20.3 Prepare to use the Computer Vision SDK . . . 76
20.4 View the images you will analyze . . . 76
20.5 Analyze an image to suggest a caption . . . 76
20.6 Get suggested tags for an image . . . 78
20.7 Get image categories . . . 79
20.8 Get brands in an image . . . 80
20.9 Detect and locate objects in an image . . . 81
20.10 Get moderation ratings for an image . . . 82
20.11 Generate a thumbnail image . . . 82
20.12 More information . . . 83
20.13 For more information about using the **Computer Vision** service, see the Computer Vision documentation. . . . 83
20.14 lab: title: 'Analyze Video with Video Indexer' module: 'Module 8 - Getting Started with Computer Vision' . . . 83

21 Analyze Video with Video Indexer 83
21.1 Clone the repository for this course . . . 83
21.2 Upload a video to Video Indexer . . . 84
21.3 Review video insights . . . 84
21.4 Search for insights . . . 85
21.5 Edit insights . . . 86
21.6 Use Video Indexer widgets . . . 87
21.7 Use the Video Indexer REST API . . . 88
21.7.1 Get your API details . . . 88
21.7.2 Use the REST API . . . 88
21.8 More information . . . 89
21.9 For more information about **Video Indexer**, see the Video Indexer documentation. . . . 89
21.10 lab: title: 'Classify Images with Custom Vision' module: 'Module 9 - Developing Custom Vision Solutions' . . . 89

22 Classify Images with Custom Vision 89
22.1 Clone the repository for this course . . . 89
22.2 Create Custom Vision resources . . . 89
22.3 Create a Custom Vision project . . . 90
22.4 Test the model . . . 91
22.5 View the project settings . . . 91
22.6 Use the *training* API . . . 92
22.7 Publish the image classification model . . . 93
22.8 Use the image classifier from a client application . . . 93
22.9 More information . . . 94
22.10 For more information about image classification with the Custom Vision service, see the Custom Vision documentation. . . . 94
22.11 lab: title: 'Detect Objects in Images with Custom Vision' module: 'Module 9 - Developing Custom Vision Solutions' . . . 94

23 Detect Objects in Images with Custom Vision 94
23.1 Clone the repository for this course . . . 94
23.2 Create Custom Vision resources . . . 94
23.3 Create a Custom Vision project . . . 94
23.4 Add and tag images . . . 95
23.5 Use the Training API to upload images . . . 96
23.6 Train and test a model . . . 97
23.7 Publish the object detection model . . . 97
23.8 Use the image classifier from a client application . . . 98

23.9 More information 99
23.10 For more information about object detection with the Custom Vision service, see the Custom Vision documentation. 99
23.11 lab: title: 'Detect, Analyze, and Recognize Faces' module: 'Module 10 - Detecting, Analyzing, and Recognizing Faces' 99

24 Detect, Analyze, and Recognize Faces 99
24.1 Clone the repository for this course 99
24.2 Provision a Cognitive Services resource 99
24.3 Prepare to use the Computer Vision SDK 99
24.4 View the image you will analyze 100
24.5 Detect faces in an image 100
24.6 Prepare to use the Face SDK 102
24.7 Detect and analyze faces 103
24.8 Compare faces 105
24.9 Train a facial recognition model 108
24.10 Recognize faces 110
24.11 Verify a face 112
24.12 More information 113
24.13 To learn more about the **Face** service, see the Face documentation. 113
24.14 lab: title: 'Read Text in Images' module: 'Module 11 - Reading Text in Images and Documents' 113

25 Read Text in Images 113
25.1 Clone the repository for this course 114
25.2 Provision a Cognitive Services resource 114
25.3 Prepare to use the Computer Vision SDK 114
25.4 Use the OCR API 115
25.5 Use the Read API 117
25.6 Read handwritten text 118
25.7 More information 118
25.8 For more information about using the **Computer Vision** service to read text, see the Computer Vision documentation. 118
25.9 lab: title: 'Extract Data from Forms' module: 'Module 11 - Reading Text in Images and Documents' 118

26 Extract Data from Forms 118
26.1 Clone the repository for this course 119
26.2 Create a Form Recognizer resource 119
26.3 Gather documents for training 120
26.4 Train a model *without* labels 122
26.5 Test the model created without labels 123
26.6 Train a model *with* labels using the client library 123
26.7 Test the model created with labels 124
26.8 More information 125
26.9 For more information about the Form Recognizer service, see the Form Recognizer documentation. 125
26.10 lab: title: 'Create an Azure Cognitive Search solution' module: 'Module 12 - Creating a Knowledge Mining Solution' 125

27 Create an Azure Cognitive Search solution 125
27.1 Clone the repository for this course 125
27.2 Create Azure resources 125
27.2.1 Create an Azure Cognitive Search resource 125
27.2.2 Create a Cognitive Services resource 126
27.2.3 Create a storage account 126
27.3 Upload Documents to Azure Storage 126
27.4 Index the documents 127
27.5 Search the index 128
27.6 Explore and modify definitions of search components 128
27.6.1 Get the endpoint and key for your Azure Cognitive Search resource 129
27.6.2 Review and modify the skillset 129
27.6.3 Review and modify the index 130
27.6.4 Review and modify the indexer 130

27.6.5 Use the REST API to update the search solution . . . 131
27.6.6 Query the modified index . . . 131
27.7 Create a search client application . . . 131
27.7.1 Get the endpoint and keys for your search resource . . . 131
27.7.2 Prepare to use the Azure Cognitive Search SDK . . . 131
27.7.3 Explore code to search an index . . . 132
27.7.4 Explore code to render search results . . . 132
27.7.5 Run the web app . . . 132
27.8 More information . . . 133
27.9 To learn more about Azure Cognitive Search, see the Azure Cognitive Search documentation. . . 133
27.10lab: title: 'Create a Custom Skill for Azure Cognitive Search' module: 'Module 12 - Creating a Knowledge Mining Solution' . . . 133

28 Create a Custom Skill for Azure Cognitive Search 133
28.1 Clone the repository for this course . . . 133
28.2 Create Azure resources . . . 134
28.3 Create a search solution . . . 135
28.4 Search the index . . . 136
28.5 Create an Azure Function for a custom skill . . . 136
28.5.1 **C#** . . . 136
28.5.2 **Python** . . . 136
28.5.3 **C#** . . . 137
28.6 **Python** . . . 139
28.6.1 **C#** . . . 141
28.6.2 **Python** . . . 141
28.7 Test the function . . . 141
28.8 Add the custom skill to the search solution . . . 142
28.9 Search the index . . . 143
28.10More information . . . 144
28.11To learn more about creating custom skills for Azure Cognitive Search, see the Azure Cognitive Search documentation. . . . 144
28.12lab: title: 'Create a Knowledge Store with Azure Cognitive Search' module: 'Module 12 - Creating a Knowledge Mining Solution' . . . 144

29 Create a Knowledge Store with Azure Cognitive Search 144
29.1 Clone the repository for this course . . . 144
29.2 Create Azure resources . . . 144
29.3 Create a search solution . . . 145
29.3.1 Prepare JSON for REST operations . . . 145
29.3.2 Submit REST requests . . . 146
29.4 View the knowledge store . . . 147
29.4.1 View object projections . . . 147
29.4.2 View file projections . . . 148
29.4.3 View table projections . . . 148
29.5 More information . . . 148

1 AI-102: AI Engineer

This repo contains the lab instructions and files used in Microsoft Official Curriculum course AI-102.

- **Are you a MCT?** - Have a look at our GitHub User Guide for MCTs
- **Need to manually build the lab instructions?** - Instructions are available in the MicrosoftLearning/Docker-Build repository

1.1 What are we doing?

- To support this course, we will need to make frequent updates to the course content to keep it current with the Azure services used in the course. We are publishing the lab instructions and lab files on GitHub to allow for open contributions between the course authors and MCTs to keep the content current with changes in the Azure platform.

- We hope that this brings a sense of collaboration to the labs like we've never had before - when Azure changes and you find it first during a live delivery, go ahead and make an enhancement right in the lab source. Help your fellow MCTs.

1.2 How should I use these files relative to the released MOC files?

- The instructor materials are still going to be your primary source for teaching the course content.
- These files on GitHub are designed to be used in the course labs, using the hosted lab environment.
- We recommend that for every delivery, trainers check GitHub for any changes that may have been made to support the latest Azure services, and get the latest files for their delivery.

1.3 What about changes to the student handbook?

- We will review the student handbook on a quarterly basis and update through the normal MOC release channels as needed.

1.4 How do I contribute?

- Any MCT can submit a pull request to the code or content in the GitHub repo, Microsoft and the course author will triage and include content and lab code changes as needed.
- You can submit bugs, changes, improvement and ideas. Find a new Azure feature before we have? Submit a new demo!

1.5 title: Online Hosted Instructions permalink: index.html layout: home

2 Content Directory

Hyperlinks to each of the lab exercises and demos are listed below.

2.1 Labs

{% assign labs = site.pages | where_exp:"page", "page.url contains '/Instructions'" %} | Module | Lab | | --- | --- | {% for activity in labs %}| {{ activity.lab.module }} | [{{ activity.lab.title }}{% if activity.lab.type %} - {{ activity.lab.type }}{% endif %}](/home/ll/Azure_clone/Azure_new/AI-102-AIEngineer/{{ site.github.url }}{{ activity.url }}) | {% endfor %}

3 The MIT License (MIT)

3.2 lab: title: ’Lab Environment Setup’ module: ’Setup’

4 Lab Environment Setup

These exercises are designed to be completed in a hosted lab environment. If you want to complete them on your own computer, you can do so by installing the following software. You may experience unexpected dialogs and behavior when using your own environment. Due to the wide range of possible local configurations, the course team cannot support issues you may encounter in your own environment.

> **Note**: The instructions below are for a Windows 10 computer. You can also use Linux or MacOS. You may need to adapt the lab instructions for your chosen OS.

4.0.1 Base Operating System (Windows 10)

4.0.1.1 Windows 10

Install Windows 10 and apply all updates.

4.0.1.2 Edge

Install Edge (Chromium)

4.0.2 .NET Core SDK

1. Download and install from https://dotnet.microsoft.com/download (download .NET Core SDK - not just the runtime)

4.0.3 C++ Redistributable

1. Download and install the Visual C++ Redistributable (x64) from https://aka.ms/vs/16/release/vc_redist.x64.exe.

4.0.4 Node.JS

1. Download the latest LTS version from https://nodejs.org/en/download/
2. Install using the default options

4.0.5 Python (and required packages)

1. Download version 3.8 from https://docs.conda.io/en/latest/miniconda.html
2. Run setup to install - **Important**: Select the options to add Miniconda to the PATH variable and to register Miniconda as the default Python environment.
3. After installation, open the Anaconda prompt and enter the following commands to install packages:

```
pip install flask requests python-dotenv pylint matplotlib pillow
pip install --upgrade numpy
```

4.0.6 Azure CLI

1. Download from https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest
2. Install using the default options

4.0.7 Git

1. Download and install from https://git-scm.com/download.html, using the default options

4.0.8 Visual Studio Code (and extensions)

1. Download from https://code.visualstudio.com/Download
2. Install using the default options
3. After installation, start Visual Studio Code and on the **Extensions** tab (CTRL+SHIFT+X), search for and install the following extensions from Microsoft:
   - Python
   - C#
   - Azure Functions
   - PowerShell

4.0.9 Bot Framework Emulator

Follow the instructions at https://github.com/Microsoft/BotFramework-Emulator/blob/master/README.md to download and install the latest stable version of the Bot Framework Emulator for your operating system.

4.0.10 Bot Framework Composer

4.1 Install from https://docs.microsoft.com/en-us/composer/install-composer.

4.2 lab: title: 'Enable Resource Providers' module: 'Setup'

5 Enable Resource Providers

There are some resource providers that must be registered in your Azure subscription. Follow these steps to ensure that they're registered.

1. Sign into the Azure portal at `https://portal.azure.com` using the Microsoft credentials associated with your subscription.
2. On the **Home** page, select **Subscriptions** (or expand the menu, select **All Services**, and in the **All** category, select **Subscriptions**).
3. Select your Azure subscription (if you have multiple subscriptions, select the one you created by redeeming your Azure Pass).
4. In the blade for your subscription, in the pane on the left, in the **Settings** section, select **Resource providers**.
5. In the list of resource providers, ensure the following providers are registered (if not, select them and click **register**):
   - Microsoft.BotService
   - Microsoft.Web
   - Microsoft.ManagedIdentity
   - Microsoft.Search
   - Microsoft.Storage
   - Microsoft.CognitiveServices
   - Microsoft.AlertsManagement
   - microsoft.insights
   - Microsoft.KeyVault
   - Microsoft.ContainerInstance

5.1 lab: title: 'Get Started with Cognitive Services' module: 'Module 2 - Developing AI Apps with Cognitive Services'

6 Get Started with Cognitive Services

In this exercise, you'll get started with Cognitive Services by creating a **Cognitive Services** resource in your Azure subscription and using it from a client application. The goal of the exercise is not to gain expertise in any particular service, but rather to become familiar with a general pattern for provisioning and working with cognitive services as a developer.

6.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

6.2 Provision a Cognitive Services resource

Azure Cognitive Services are cloud-based services that encapsulate artificial intelligence capabilities you can incorporate into your applications. You can provision individual cognitive services resources for specific APIs (for example, **Text Analytics** or **Computer Vision**), or you can provision a general **Cognitive Services** resource that provides access to multiple cognitive services APIs through a single endpoint and key. In this case, you'll use a single **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
    - **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)*
    - **Region**: *Choose any available region*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. Go to the resource and view its **Keys and Endpoint** page. This page contains the information that you will need to connect to your resource and use it from applications you develop. Specifically:
    - An HTTP *endpoint* to which client applications can send requests.
    - Two *keys* that can be used for authentication (client applications can use either key to authenticate).
    - The *location* where the resource is hosted. This is required for requests to some (but not all) APIs.

6.3 Use a REST Interface

The cognitive services APIs are REST-based, so you can consume them by submitting JSON requests over HTTP. In this example, you'll explore a console application that uses the **Text Analytics** REST API to perform language detection; but the basic principle is the same for all of the APIs supported by the Cognitive Services resource.

> **Note**: In this exercise, you can choose to use the REST API from 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 **01-getting-started** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. View the contents of the **rest-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 reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.
3. Note that the **rest-client** folder contains a code file for the client application:
    - **C#**: Program.cs
    - **Python**: rest-client.py

Open the code file and review the code it contains, noting the following details:

- Various namespaces are imported to enable HTTP communication
- Code in the **Main** function retrieves the endpoint and key for your cognitive services resource - these will be used to send REST requests to the Text Analytics service.
- The program accepts user input, and uses the **GetLanguage** function to call the Text Analytics language detection REST API for your cognitive services endpoint to detect the language of the text that was entered.
- The request sent to the API consists of a JSON object containing the input data - in this case, a collection of **document** objects, each of which has an **id** and **text**.
- The key for your service is included in the request header to authenticate your client application.
- The response from the service is a JSON object, which the client application can parse.

4. Right-click the **rest-client** folder and open an integrated terminal. Then enter the following language-specific command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python rest-client.py
   ```

5. When prompted, enter some text and review the language that is detected by the service, which is returned in the JSON response. For example, try entering ”Hello”, ”Bonjour”, and ”Hola”.

6. When you have finished testing the application, enter ”quit” to stop the program.

6.4 Use an SDK

You can write code that consumes cognitive services REST APIs directly, but there are software development kits (SDKs) for many popular programming languages, including Microsoft C#, Python, and Node.js. Using an SDK can greatly simplify development of applications that consume cognitive services.

1. In Visual Studio Code, in the **Explorer** pane, in the **01-getting-started** folder, expand the **C-Sharp** or **Python** folder depending on your language preference.

2. Right-click the **sdk-client** folder and open an integrated terminal. Then install the Text Analytics SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Azure.AI.TextAnalytics --version 5.0.0
   ```

   Python

   ```
   pip install azure-ai-textanalytics==5.0.0
   ```

3. View the contents of the **sdk-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 reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.

4. Note that the **sdk-client** folder contains a code file for the client application:

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

   Open the code file and review the code it contains, noting the following details:

   - The namespace for the SDK you installed is imported
   - Code in the **Main** function retrieves the endpoint and key for your cognitive services resource - these will be used with the SDK to create a client for the Text Analytics service.
   - The **GetLanguage** function uses the SDK to create a client for the service, and then uses the client to detect the language of the text that was entered.

5. Return to the integrated terminal for the **sdk-client** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python sdk-client.py
   ```

6. When prompted, enter some text and review the language that is detected by the service. For example, try entering ”Goodbye”, ”Au revoir”, and ”Hasta la vista”.

7. When you have finished testing the application, enter ”quit” to stop the program.

   Note: Some languages that require Unicode character sets may not be recognized in this simple console application.

6.5 More information

6.6 For more information about Azure Cognitive Services, see the Cognitive Services documentation.

6.7 lab: title: ’Manage Cognitive Services Security’ module: ’Module 2 - Developing AI Apps with Cognitive Services’

7 Manage Cognitive Services Security

Security is a critical consideration for any application, and as a developer you should ensure that access to resources such as cognitive services is restricted to only those who require it.

Access to cognitive services is typically controlled through authentication keys, which are generated when you initially create a cognitive services resource.

7.1 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/MicrosoftLe` 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**.

7.2 Provision a Cognitive Services resource

If you don’t already have one in your subscription, you’ll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.

7.3 Manage authentication keys

When you created your cognitive services resource, two authentication keys were generated. You can manage these in the Azure portal or by using the Azure command line interface (CLI).

1. In the Azure portal, go to your cognitive services resource and view its **Keys and Endpoint** page. This page contains the information that you will need to connect to your resource and use it from applications you develop. Specifically:
   - An HTTP *endpoint* to which client applications can send requests.
   - Two *keys* that can be used for authentication (client applications can use either of the keys. A common practice is to use one for development, and another for production. You can easily regenerate the development key after developers have finished their work to prevent continued access).
   - The *location* where the resource is hosted. This is required for requests to some (but not all) APIs.
2. In Visual Studio Code, right-click the **02-cognitive-security** folder and open an integrated terminal. Then enter the following command to sign into your Azure subscription by using the Azure CLI.

   ```
   az login
   ```

   A web browser tab will open and prompt you to sign into Azure. Do so, and then close the browser tab and return to Visual Studio Code.

   > **Tip**: If you have multiple subscriptions, you'll need to ensure that you are working in the one that contains your cognitive services resource. Use this command to determine your current subscription - its unique ID is the **id** value in the JSON that gets returned.
   >
   > ```
   > az account show
   > ```
   >
   > If you need to change the subscription, run this command, changing *<Your_Subscription_Id>* to the correct subscription ID.
   >
   > ```
   > az account set --subscription <Your_Subscription_Id>
   > ```
   >
   > Alternatively, you can explicitly specify the subscription ID as a *--subscription* parameter in each Azure CLI command that follows.

3. Now you can use the following command to get the list of cognitive services keys, replacing *<resourceName>* with the name of your cognitive services resource, and *<resourceGroup>* with the name of the resource group in which you created it.

   ```
   az cognitiveservices account keys list --name <resourceName> --resource-group <resourceGroup>
   ```

The command returns a list of the keys for your cognitive services resource - there are two keys, named **key1** and **key2**.

4. To test your cognitive service, you can use **curl** - a command line tool for HTTP requests. In the **02-cognitive-security** folder, open **rest-test.cmd** and edit the **curl** command it contains (shown below), replacing *<yourEndpoint>* and *<yourKey>* with your endpoint URI and **Key1** key to use the Text Analytics API in your cognitive services resource.

   ```
   curl -X POST "<yourEndpoint>/text/analytics/v3.0/languages?" -H "Content-Type: application/json" -
   ```

5. Save your changes, and then in the integrated terminal for the **02-cognitive-security** folder, run the following command:

   ```
   rest-test
   ```

The command returns a JSON document containing information about the language detected in the input data (which should be English).

6. If a key becomes compromised, or the developers who have it no longer require access, you can regenerate it in the portal or by using the Azure CLI. Run the following command to regenerate your **key1** key (replacing *<resourceName>* and *<resourceGroup>* for your resource).

   ```
   az cognitiveservices account keys regenerate --name <resourceName> --resource-group <resourceGroup
   ```

The list of keys for your cognitive services resource is returned - note that **key1** has changed since you last retrieved them.

7. Re-run the **rest-test** command with the old key (you can use the ^ key to cycle through previous commands), and verify that it now fails.

8. Edit the *curl* command in **rest-test.cmd** replacing the key with the new **key1** value, and save the changes. Then rerun the **rest-test** command and verify that it succeeds.

   Tip: In this exercise, you used the full names of Azure CLI parameters, such as **--resource-group**. You can also use shorter alternatives, such as **-g**, to make your commands less verbose (but a little harder to understand). The Cognitive Services CLI command reference lists the parameter options for each cognitive services CLI command.

7.4 Secure key access with Azure Key Vault

You can develop applications that consume cognitive services by using a key for authentication. However, this means that the application code must be able to obtain the key. One option is to store the key in an environment variable or a configuration file where the application is deployed, but this approach leaves the key vulnerable to unauthorized access. A better approach when developing applications on Azure is to store the key securely in Azure Key Vault, and provide access to the key through a *managed identity* (in other words, a user account used by the application itself).

7.4.1 Create a key vault and add a secret

First, you need to create a key vault and add a *secret* for the cognitive services key.

1. Make a note of the **key1** value for your cognitive services resource (or copy it to the clipboard).
2. In the Azure portal, on the **Home** page, select the **Create a resource** button, search for *Key Vault*, and create a **Key Vault** resource with the following settings:
   - **Subscription**: *Your Azure subscription*
   - **Resource group**: *The same resource group as your cognitive service resource*
   - **Key vault name**: *Enter a unique name*
   - **Region**: *The same region as your cognitive service resource*
   - **Pricing tier**: Standard
3. Wait for deployment to complete and then go to your key vault resource.
4. In the left navigation pane, select **Secrets** (in the Settings section).
5. Select **+ Generate/Import** and add a new secret with the following settings :
   - **Upload options**: Manual
   - **Name**: Cognitive-Services-Key *(it's important to match this exactly, because later you'll run code that retrieves the secret based on this name)*
   - **Value**: *Your **key1** cognitive services key*

7.4.2 Create a service principal

To access the secret in the key vault, your application must use a service principal that has access to the secret. You'll use the Azure command line interface (CLI) to create the service principal and grant access to the secret in Azure Vault.

1. Return to Visual Studio Code, and in the interactive terminal for the **02-cognitive-security** folder, run the following Azure CLI command, replacing *<spName>* with a suitable name for an application identity (for example, *ai-app*). Also replace *<subscriptionId>* and *<resourceGroup>* with the correct values for your subscription ID and the resource group containing your cognitive services and key vault resources:

   > **Tip**: If you are unsure of your subscription ID, use the **az account show** command to retrieve your subscription information - the subscription ID is the **id** attribute in the output.

   ```
   az ad sp create-for-rbac -n "https://<spName>" --role owner --scopes subscriptions/<subscriptionId
   ```

The output of this command includes information about your new service principal. It should look similar to this:

```
{
    "appId": "abcd12345efghi67890jklmn",
    "displayName": "ai-app",
    "name": "https://ai-app",
    "password": "1a2b3c4d5e6f7g8h9i0j",
    "tenant": "1234abcd5678fghi90jklm"
}
```

Make a note of the **appId**, **password**, and **tenant** values - you will need them later (if you close this terminal, you won't be able to retrieve the password; so it's important to note the values now - you can paste the output into a new text file in Visual Studio Code to ensure you can find the values you need later!)

2. To assign permission for your new service principal to access secrets in your Key Vault, run the following Azure CLI command, replacing *<keyVaultName>* with the name of your Azure Key Vault resource and *<spName>* with the same value you provided when creating the service principal.

   ```
   az keyvault set-policy -n <keyVaultName> --spn "https://<spName>" --secret-permissions get list
   ```

7.4.3 Use the service principal in an application

Now you're ready to use the service principal identity in an application, so it can access the secret congitive services key in your key vault and use it to connect to your cognitive services resource.

> **Note**: In this exercise, we'll store the service principal credentials in the application configuration and use them to authenticate a **ClientSecretCredential** identity in your application code. This is fine for development and testing, but in a real production application, an administrator would assign a *managed identity* to the application so that it uses the service principal identity to access resources, without caching or storing the password.

1. In Visual Studio Code, expand the **02-cognitive-security** folder and the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **keyvault-client** folder and open an integrated terminal. Then install the packages you will need to use Azure Key Vault and the Text Analytics API in your cognitive services resource by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Azure.AI.TextAnalytics --version 5.0.0
   dotnet add package Azure.Identity --version 1.3.0
   dotnet add package Azure.Security.KeyVault.Secrets --version 4.2.0-beta.3
   ```

   Python

   ```
   pip install azure-ai-textanalytics==5.0.0
   pip install azure-identity==1.5.0
   pip install azure-keyvault-secrets==4.2.0
   ```

3. View the contents of the **keyvault-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 reflect the following settings:

   - The **endpoint** for your Cognitive Services resource
   - The name of your **Azure Key Vault** resource
   - The **tenant** for your service principal
   - The **appId** for your service principal
   - The **password** for your service principal

   Save your changes.

4. Note that the **keyvault-client** folder contains a code file for the client application:

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

   Open the code file and review the code it contains, noting the following details:

   - The namespace for the SDK you installed is imported
   - Code in the **Main** function retrieves the application configuration settings, and then it uses the service principal credentials to get the cognitive services key from the key vault.
   - The **GetLanguage** function uses the SDK to create a client for the service, and then uses the client to detect the language of the text that was entered.

5. Return to the integrated terminal for the **keyvault-client** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python keyvault-client.py
```

6. When prompted, enter some text and review the language that is detected by the service. For example, try entering "Hello", "Bonjour", and "Hola".
7. When you have finished testing the application, enter "quit" to stop the program.

7.5 More information

7.6 For more information about securing cognitive services, see the Cognitive Services security documentation.

7.7 lab: title: 'Monitor Cognitive Services' module: 'Module 2 - Developing AI Apps with Cognitive Services'

8 Monitor Cognitive Services

Azure Cognitive Services can be a critical part of an overall application infrastructure. It's important to be able to monitor activity and get alerted to issues that may need attention.

8.1 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/MicrosoftLe` 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**.

8.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select any required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. Make a note of the endpoint URI - you will need it later.

8.3 Configure an alert

Let's start monitoring by defining an alert rule so you can detect activity in your cognitive services resource.

1. In the Azure portal, go to your cognitive services resource and view its **Alerts** page (in the **Monitoring** section).

2. Select + **New alert rule**
3. In the **Create alert rule** page, under **Scope**, verify that the your cognitive services resource is listed.
4. Under **Condition**, click **Add Condition**, and view the **Configure signal logic** pane that appears on the right, where you can select a signal type to monitor.
5. In the **signal type** list, select **Activity Log**, and then in the filtered list, select **List Keys**.
6. Review the activity over the past 6 hours, and then select **Done**.
7. Back in the **Create alert rule** page, under **Actions**, note that you can specify an *action group*. This enables you to configure automated actions when an alert is fired - for example, sending an email notification. We won't do that in this exercise; but it can be useful to do this in a production environment.
8. In the **Alert Details** section, set the **Alert rule name** to **Key List Alert**, and click **Create alert rule**. Wait for the alert rule to be created.
9. In Visual Studio Code, right-click the **03-monitor** folder and open an integrated terminal. Then enter the following command to sign into your Azure subscription by using the Azure CLI.

   ```
   az login
   ```

   A web browser tab will open and prompt you to sign into Azure. Do so, and then close the browser tab and return to Visual Studio Code.

   > **Tip**: If you have multiple subscriptions, you'll need to ensure that you are working in the one that contains your cognitive services resource. Use this command to determine your current subscription.
   >
   > ```
   > az account show
   > ```
   >
   > If you need to change the subscription, run this command, changing *<subscriptionName>* to the correct subscription name.
   >
   > ```
   > az account set --subscription <subscriptionName>
   > ```

10. Now you can use the following command to get the list of cognitive services keys, replacing *<resourceName>* with the name of your cognitive services resource, and *<resourceGroup>* with the name of the resource group in which you created it.

    ```
    az cognitiveservices account keys list --name <resourceName> --resource-group <resourceGroup>
    ```

The command returns a list of the keys for your cognitive services resource.

11. Switch back to the browser containing the Azure portal, and refresh your **Alert page**. You should see a **Sev 4** alert listed in the table (if not, wait up to five minutes and refresh again).
12. Select the alert to see its details.

8.4 Visualize a metric

As well as defining alerts, you can view metrics for your cognitive services resource to monitor its utilization.

1. In the Azure portal, in the page for your cognitive services resource, select **Metrics** (in the **Monitoring** section).
2. If there is no existing chart, select + **New chart**. Then in the **Metric** list, review the possible metrics you can visualize and select **Total Calls**.
3. In the **Aggregation** list, select **Count**. This will enable you to monitor the total calls to you Cognitive Service resource; which is useful in determining how much the service is being used over a period of time.
4. To generate some requests to your cognitive service, you will use **curl** - a command line tool for HTTP requests. In Visual Studio Code, in the **03-monitor** folder, open **rest-test.cmd** and edit the **curl** command it contains (shown below), replacing *<yourEndpoint>* and *<yourKey>* with your endpoint URI and **Key1** key to use the Text Analytics API in your cognitive services resource.

   ```
   curl -X POST "<yourEndpoint>/text/analytics/v3.0/languages?" -H "Content-Type: application/json" -
   ```

5. Save your changes, and then in the integrated terminal for the **03-monitor** folder, run the following command:

   ```
   rest-test
   ```

The command returns a JSON document containing information about the language detected in the input data (which should be English).

6. Re-run the **rest-test** command multiple times to generate some call activity (you can use the ^ key to cycle through previous commands).
7. Return to the **Metrics** page in the Azure portal and refresh the **Total Calls** count chart. It may take a few minutes for the calls you made using *curl* to be reflected in the chart - keep refreshing the chart until it updates to include them.

8.5 More information

8.6 One of the options for monitoring cognitive services is to use *diagnostic logging*. Once enabled, diagnostic logging captures rich information about your cognitive services resource usage, and can be a useful monitoring and debugging tool. It can take over an hour after setting up diagnostic logging to generate any information, which is why we haven't explored it in this exercise; but you can learn more about it in the Cognitive Services documentation.

8.7 lab: title: 'Use a Cognitive Services Container' module: 'Module 2 - Developing AI Apps with Cognitive Services'

9 Use a Cognitive Services Container

Using cognitive services hosted in Azure enables application developers to focus on the infrastructure for their own code while benefiting from scalable services that are managed by Microsoft. However, in many scenarios, organizations require more control over their service infrastructure and the data that is passed between services.

Many of the cognitive services APIs can be packaged and deployed in a *container*, enabling organizations to host cognitive services in their own infrastructure; for example in local Docker servers, Azure Container Instances, or Azure Kubernetes Services clusters. Containerized cognitive services need to communicate with an Azure-based cognitive services account to support billing; but application data is not passed to the back-end service, and organizations have greater control over the deployment configuration of their containers, enabling custom solutions for authentication, scalability, and other considerations.

9.1 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/MicrosoftLe` 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**.

9.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0

3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

9.3 Deploy and run a Text Analytics container

Many commonly used cognitive services APIs are available in container images. For a full list, check out the cognitive services documentation. In this exercise, you'll use the container image for the Text Analytics *language detection* API; but the principles are the same for all of the available images.

1. In the Azure portal, on the **Home** page, select the **Create a resource** button, search for *container instances*, and create a **Container Instances** resource with the following settings:

    - **Basics**:
        - **Subscription**: *Your Azure subscription*
        - **Resource group**: *Choose the resource group containing your cognitive services resource*
        - **Container name**: *Enter a unique name*
        - **Region**: *Choose any available region*
        - **Image source**: Docker Hub or other Registry
        - **Image type**: Public
        - **Image**: `mcr.microsoft.com/azure-cognitive-services/textanalytics/language:1.1.012840001-amd6`
        - **OS type**: Linux
        - **Size**: 1 vcpu, 4 GB memory
    - **Networking**:
        - **Networking type**: Public
        - **DNS name label**: *Enter a unique name for the container endpoint*
        - **Ports**: *Change the TCP port from 80 to **5000***
    - **Advanced**:
        - **Restart policy**: On failure
        - **Environment variables**:

Mark as secure	Key	Value
Yes	ApiKey	*Either key for your cognitive services resource*
Yes	Billing	*The endpoint URI for your cognitive services resource*
No	Eula	accept

        - **Command override**: [ ]
    - **Tags**:
        - *Don't add any tags*

2. Wait for deployment to complete, and then go to the deployed resource.

3. Observe the following properties of your container instance resource on its **Overview** page:

    - **Status**: This should be *Running*.
    - **IP Address**: This is the public IP address you can use to access your container instances.
    - **FQDN**: This is the *fully-qualified domain name* of the container instances resource, you can use this to access the container instances instead of the IP address.

    Note: In this exercise, you've deployed the cognitive services container image for text translation to an Azure Container Instances (ACI) resource. You can use a similar approach to deploy it to a *Docker* host on your own computer or network by running the following command (on a single line) to deploy the language detection container to your local Docker instance, replacing *<yourEndpoint>* and *<yourKey>* with your endpoint URI and either of the keys for your cognitive services resource.

    ```
    docker run --rm -it -p 5000:5000 --memory 4g --cpus 1 mcr.microsoft.com/azure-cognitive-servic
    ```

    The command will look for the image on your local machine, and if it doesn't find it there it will pull it from the *mcr.microsoft.com* image registry and deploy it to your Docker instance. When deployment is complete, the container will start and listen for incoming requests on port 5000.

9.4 Use the container

1. In Visual Studio Code, in the **04-containers** folder, open **rest-test.cmd** and edit the **curl** command it contains (shown below), replacing *<your_ACI_IP_address_or_FQDN>* with the IP address or FQDN for your container.

   ```
   curl -X POST "http://<your_ACI_IP_address_or_FQDN>:5000/text/analytics/v3.0/languages?" -H "Conten
   ```

2. Save your changes to the script. Note that you do not need to specify the cognitive services endpoint or key - the request is processed by the containerized service. The container in turn communicates periodically with the service in Azure to report usage for billing, but does not send request data.

3. Right-click the **04-containers** folder and open an integrated terminal. Then enter the following command to run the script:

   ```
   rest-test
   ```

4. Verify that the command returns a JSON document containing information about the language detected in the two input documents (which should be English and French).

9.5 Clean Up

If you've finished experimenting with your container instance, you should delete it.

1. In the Azure portal, open the resource group where you created your resources for this exercise.
2. Select the container instance resource and delete it.

9.6 More information

9.7 For more information about containerizing cognitive services, see the Cognitive Services containers documentation.

9.8 lab: title: 'Analyze Text' module: 'Module 3 - Getting Started with Natural Language Processing'

10 Analyze Text

The **Text Analytics API** is a cognitive service that 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 Text Analytics API, 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.

10.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.

2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

10.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

10.3 Prepare to use the Text Analytics SDK

In this exercise, you'll complete a partially implemented client application that uses the Text Analysis SDK to analyze hotel reviews.

> **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 **05-analyze-text** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.

2. Right-click the **text-analysis** folder and open an integrated terminal. Then install the Text Analytics SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Azure.AI.TextAnalytics --version 5.0.0
   ```

   Python

   ```
   pip install azure-ai-textanalytics==5.0.0
   ```

3. View the contents of the **text-analysis** folder, and note that it contains a file for configuration settings:

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

   Open the configuration file and update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.

4. Note that the **text-analysis** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: text-analysis.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 Text Analytics SDK:

   C#

   ```
   // import namespaces
   using Azure;
   using Azure.AI.TextAnalytics;
   ```

   Python

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

5. In the **Main** function, note that code to load the cognitive services 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:

   C#

```
// Create client using endpoint and key
AzureKeyCredential credentials = new AzureKeyCredential(cogSvcKey);
Uri endpoint = new Uri(cogSvcEndpoint);
TextAnalyticsClient CogClient = new TextAnalyticsClient(endpoint, credentials);
```

Python

```
# Create client using endpoint and key
credential = AzureKeyCredential(cog_key)
cog_client = TextAnalyticsClient(endpoint=cog_endpoint, credential=credential)
```

6. Save your changes and return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

7. 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 procedure.

10.4 Detect language

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

1. In the **Main** function for your program, find the comment **Get language**. Then, under this comment, add the code necessary to detect the language in each review document:

   C#

   ```
   // Get language
   DetectedLanguage detectedLanguage = CogClient.DetectLanguage(text);
   Console.WriteLine($"\nLanguage: {detectedLanguage.Name}");
   ```

   Python

   ```
   # Get language
   detectedLanguage = cog_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 and return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

3. Observe the output, noting that this time the language for each review is identified.

10.5 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 **Main** function for your program, find the comment **Get sentiment**. Then, under this comment, add the code necessary to detect the sentiment of each review document:

   C#

   ```
   // Get sentiment
   DocumentSentiment sentimentAnalysis = CogClient.AnalyzeSentiment(text);
   Console.WriteLine($"\nSentiment: {sentimentAnalysis.Sentiment}");
   ```

   Python

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

2. Save your changes and return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

3. Observe the output, noting that the sentiment of the reviews is detected.

10.6 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 **Main** function for your program, find the comment **Get key phrases**. Then, under this comment, add the code necessary to detect the key phrases in each review document:

   C#

   ```
   // Get key phrases
   KeyPhraseCollection phrases = CogClient.ExtractKeyPhrases(text);
   if (phrases.Count > 0)
   {
       Console.WriteLine("\nKey Phrases:");
       foreach(string phrase in phrases)
       {
           Console.WriteLine($"\t{phrase}");
       }
   }
   ```

   Python

   ```
   # Get key phrases
   phrases = cog_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 return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

3. Observe the output, noting that each document contains key phrases that give some insights into what the review is about.

10.7 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 **Main** function for your program, find the comment **Get entities**. Then, under this comment, add the code necessary to identify entities that are mentioned in each review:

   C#

   ```
   // Get entities
   CategorizedEntityCollection entities = CogClient.RecognizeEntities(text);
   if (entities.Count > 0)
   {
       Console.WriteLine("\nEntities:");
       foreach(CategorizedEntity entity in entities)
       {
           Console.WriteLine($"\t{entity.Text} ({entity.Category})");
       }
   }
   ```

   Python

   ```
   # Get entities
   entities = cog_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 return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

3. Observe the output, noting the entities that have been detected in the text.

10.8 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 **Main** function for your program, find the comment **Get linked entities**. Then, under this comment, add the code necessary to identify linked entities that are mentioned in each review:

   C#

   ```
   // Get linked entities
   LinkedEntityCollection linkedEntities = CogClient.RecognizeLinkedEntities(text);
   if (linkedEntities.Count > 0)
   {
       Console.WriteLine("\nLinks:");
       foreach(LinkedEntity linkedEntity in linkedEntities)
       {
           Console.WriteLine($"\t{linkedEntity.Name} ({linkedEntity.Url})");
       }
   }
   ```

   Python

   ```
   # Get linked entities
   entities = cog_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 return to the integrated terminal for the **text-analysis** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

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

3. Observe the output, noting the linked entities that are identified.

10.9 More information

10.10 For more information about using the Text Analytics service, see the Text Analytics documentation.

10.11 lab: title: 'Translate Text' module: 'Module 3 - Getting Started with Natural Language Processing'

11 Translate Text

The **Translator** service is a cognitive service that enables you to translate text between languages.

For example, suppose a travel agency wants to examine hotel reviews that have been submitted to the company's web site, standardizing on English as the language that is used for analysis. By using the Translator service, they can determine the language each review is written in, and if it is not already English, translate it from whatever source language it was written in into English.

11.1 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/MicrosoftLe` 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**.

11.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need one of the keys and the location in which the service is provisioned from this page in the next procedure.

11.3 Prepare to use the Translator service

In this exercise, you'll complete a partially implemented client application that uses the Translator REST API to translate hotel reviews.

> **Note**: You can choose to use the API from 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 **06-translate-text** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. View the contents of the **text-translation** 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 an authentication **key** for your cognitive services resource, and the **location** where it is deployed (not the endpoint) - you should copy both of these from the **keys and Endpoint** page for your cognitive services resource. Save your changes.
3. Note that the **text-translation** folder contains a code file for the client application:
   - **C#**: Program.cs
   - **Python**: text-translation.py

   Open the code file and examine the code it contains.
4. In the **Main** function, note that code to load the cognitive services key and region from the configuration file has already been provided. The endpoint for the translation service is also specified in your code.
5. Right-click the **text-translation** folder, open an integrated terminal, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python text-translation.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 currently doesn't make use of the Translator service. We'll fix that in the next procedure.

11.4 Detect language

The Translator service can automatically detect the source language of text to be translated, but it also enables you to explicitly detect the language in which text is written.

1. In your code file, find the **GetLanguage** function, which currently returns "en" for all text values.
2. In the **GetLanguage** function, under the comment **Use the Translator detect function**, add the following code to use the Translator's REST API to detect the language of the specified text, being careful not to replace the code at the end of the function that returns the language:

C#

```
// Use the Translator detect function
object[] body = new object[] { new { Text = text } };
var requestBody = JsonConvert.SerializeObject(body);
using (var client = new HttpClient())
{
    using (var request = new HttpRequestMessage())
    {
        // Build the request
        string path = "/detect?api-version=3.0";
        request.Method = HttpMethod.Post;
        request.RequestUri = new Uri(translatorEndpoint + path);
        request.Content = new StringContent(requestBody, Encoding.UTF8, "application/json");
```

```
        request.Headers.Add("Ocp-Apim-Subscription-Key", cogSvcKey);
        request.Headers.Add("Ocp-Apim-Subscription-Region", cogSvcRegion);

        // Send the request and get response
        HttpResponseMessage response = await client.SendAsync(request).ConfigureAwait(false);
        // Read response as a string
        string responseContent = await response.Content.ReadAsStringAsync();

        // Parse JSON array and get language
        JArray jsonResponse = JArray.Parse(responseContent);
        language = (string)jsonResponse[0]["language"];
    }
}
```

Python

```
# Use the Translator detect function
path = '/detect'
url = translator_endpoint + path

# Build the request
params = {
    'api-version': '3.0'
}

headers = {
'Ocp-Apim-Subscription-Key': cog_key,
'Ocp-Apim-Subscription-Region': cog_region,
'Content-type': 'application/json'
}

body = [{
    'text': text
}]

# Send the request and get response
request = requests.post(url, params=params, headers=headers, json=body)
response = request.json()

# Parse JSON array and get language
language = response[0]["language"]
```

3. Save your changes and return to the integrated terminal for the **text-translation** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python text-translation.py
   ```

4. Observe the output, noting that this time the language for each review is identified.

11.5 Translate text

Now that your application can determine the language in which reviews are written, you can use the Translator service to translate any non-English reviews into English.

1. In your code file, find the **Translate** function, which currently returns and empty string for all text values.
2. In the **Translate** function, under the comment **Use the Translator translate function**, add the following code to use the Translator's REST API to translate the specified text from its source language into English, being careful not to replace the code at the end of the function that returns the translation:

C#

```
// Use the Translator translate function
object[] body = new object[] { new { Text = text } };
var requestBody = JsonConvert.SerializeObject(body);
using (var client = new HttpClient())
{
    using (var request = new HttpRequestMessage())
    {
        // Build the request
        string path = "/translate?api-version=3.0&from=" + sourceLanguage + "&to=en" ;
        request.Method = HttpMethod.Post;
        request.RequestUri = new Uri(translatorEndpoint + path);
        request.Content = new StringContent(requestBody, Encoding.UTF8, "application/json");
        request.Headers.Add("Ocp-Apim-Subscription-Key", cogSvcKey);
        request.Headers.Add("Ocp-Apim-Subscription-Region", cogSvcRegion);

        // Send the request and get response
        HttpResponseMessage response = await client.SendAsync(request).ConfigureAwait(false);
        // Read response as a string
        string responseContent = await response.Content.ReadAsStringAsync();

        // Parse JSON array and get translation
        JArray jsonResponse = JArray.Parse(responseContent);
        translation = (string)jsonResponse[0]["translations"][0]["text"];
    }
}
```

Python

```
# Use the Translator translate function
path = '/translate'
url = translator_endpoint + path

# Build the request
params = {
    'api-version': '3.0',
    'from': source_language,
    'to': ['en']
}

headers = {
    'Ocp-Apim-Subscription-Key': cog_key,
    'Ocp-Apim-Subscription-Region': cog_region,
    'Content-type': 'application/json'
}

body = [{
    'text': text
}]

# Send the request and get response
request = requests.post(url, params=params, headers=headers, json=body)
response = request.json()

# Parse JSON array and get translation
translation = response[0]["translations"][0]["text"]
```

3. Save your changes and return to the integrated terminal for the **text-translation** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

```
python text-translation.py
```

4. Observe the output, noting that non-English reviews are translated into English.

11.6 More information

11.7 For more information about using the Translator service, see the Translator documentation.

11.8 lab: title: 'Recognize and Synthesize Speech' module: 'Module 4 - Building Speech-Enabled Applications'

12 Recognize and Synthesize Speech

The **Speech** service is an Azure cognitive 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.

Note: This exercise requires that you are using a computer with speakers/headphones. For the best experience, a microphone is also required. Some hosted virtual environments may be able to capture audio from your local microphone, but if this doesn't work (or you don't have a microphone at all), you can use a provided audio file for speech input. Follow the instructions carefully, as you'll need to choose different options depending on whether you are using a microphone or the audio file.

12.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

12.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need one of the keys and the location in which the service is provisioned from this page in the next procedure.

12.3 Prepare to use the Speech service

In this exercise, you'll complete a partially implemented client application that uses the Speech SDK to recognize and synthesize speech.

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 **07-speech** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **speaking-clock** folder and open an integrated terminal. Then install the Speech SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Microsoft.CognitiveServices.Speech --version 1.14.0
   ```

   Python

   ```
   pip install azure-cognitiveservices-speech==1.14.0
   ```

3. View the contents of the **speaking-clock** 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 an authentication **key** for your cognitive services resource, and the **location** where it is deployed. Save your changes.

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

   - **C#**: Program.cs
   - **Python**: speaking-clock.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 Speech SDK:

   C#

   ```
   // Import namespaces
   using Microsoft.CognitiveServices.Speech;
   using Microsoft.CognitiveServices.Speech.Audio;
   ```

   Python

   ```
   # Import namespaces
   import azure.cognitiveservices.speech as speech_sdk
   ```

5. In the **Main** function, note that code to load the cognitive services key and region from the configuration file has already been provided. You must use these variables to create a **SpeechConfig** for your cognitive services resource. Add the following code under the comment **Configure speech service**:

   C#

   ```
   // Configure speech service
   speechConfig = SpeechConfig.FromSubscription(cogSvcKey, cogSvcRegion);
   Console.WriteLine("Ready to use speech service in " + speechConfig.Region);
   ```

   Python

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

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

   C#

   ```
   dotnet run
   ```

   Python

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

7. If you are using C#, you can ignore any warnings about using the **await** operator in asynchronous methods - we'll fix that later. The code should display the region of the speech service resource the application will use.

12.4 Recognize speech

Now that you have a **SpeechConfig** for the speech service in your cognitive services resource, you can use the **Speech-to-text** API to recognize speech and transcribe it to text.

12.4.1 If you have a working microphone

1. In the **Main** function for your program, note that the code uses the **TranscribeCommand** function to accept spoken input.
2. In the **TranscribeCommand** function, under the comment **Configure speech recognition**, add the appropriate code below to create a **SpeechRecognizer** client that can be used to recognize and transcribe speech using the default system microphone:

   C#

   ```
   // Configure speech recognition
   using AudioConfig audioConfig = AudioConfig.FromDefaultMicrophoneInput();
   using SpeechRecognizer speechRecognizer = new SpeechRecognizer(speechConfig, audioConfig);
   Console.WriteLine("Speak now...");
   ```

   Python

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

3. Now skip ahead to the **Add code to process the transcribed command** section below.

12.4.2 Alternatively, use audio input from a file

1. In the terminal window, enter the following command to install a library that you can use to play the audio file:

   C#

   ```
   dotnet add package System.Windows.Extensions --version 4.6.0
   ```

   Python

   ```
   pip install playsound==1.2.2
   ```

2. In the code file for your program, under the existing namespace imports, add the following code to import the library you just installed:

   C#

   ```
   using System.Media;
   ```

   Python

   ```
   from playsound import playsound
   ```

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

   C#

   ```
   // Configure speech recognition
   string audioFile = "time.wav";
   SoundPlayer wavPlayer = new SoundPlayer(audioFile);
   wavPlayer.Play();
   ```

```
using AudioConfig audioConfig = AudioConfig.FromWavFileInput(audioFile);
using SpeechRecognizer speechRecognizer = new SpeechRecognizer(speechConfig, audioConfig);
```

Python

```
# Configure speech recognition
audioFile = 'time.wav'
playsound(audioFile)
audio_config = speech_sdk.AudioConfig(filename=audioFile)
speech_recognizer = speech_sdk.SpeechRecognizer(speech_config, audio_config)
```

12.4.3 Add code to process the transcribed command

1. 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:

   C#

   ```
   // Process speech input
   SpeechRecognitionResult speech = await speechRecognizer.RecognizeOnceAsync();
   if (speech.Reason == ResultReason.RecognizedSpeech)
   {
       command = speech.Text;
       Console.WriteLine(command);
   }
   else
   {
       Console.WriteLine(speech.Reason);
       if (speech.Reason == ResultReason.Canceled)
       {
           var cancellation = CancellationDetails.FromResult(speech);
           Console.WriteLine(cancellation.Reason);
           Console.WriteLine(cancellation.ErrorDetails);
       }
   }
   ```

   Python

   ```
   # 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)
   ```

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

   C#

   ```
   dotnet run
   ```

   Python

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

3. If using a microphone, speak clearly and say ”what time is it?”. The program should transcribe your spoken input and display the time (based on the local time of the computer where the code is running, which may not be the correct time where you are).

The SpeechRecognizer gives you around 5 seconds to speak. If it detects no spoken input, it produces a "No match" result.

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 key or region in the configuration file.

12.5 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.

1. In the **Main** function for your program, 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:

   C#

   ```
   // Configure speech synthesis
   using SpeechSynthesizer speechSynthesizer = new SpeechSynthesizer(speechConfig);
   ```

   Python

   ```
   # Configure speech synthesis
   speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config)
   ```

   > **Note**: *The default audio configuration uses the default system audio device for output, so you don't need to explicitly provide an* ***AudioConfig****. If you need to redirect audio output to a file, you can use an* ***AudioConfig*** *with a filepath to do so.*

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:

   C#

   ```
   // Synthesize spoken output
   SpeechSynthesisResult speak = await speechSynthesizer.SpeakTextAsync(responseText);
   if (speak.Reason != ResultReason.SynthesizingAudioCompleted)
   {
       Console.WriteLine(speak.Reason);
   }
   ```

   Python

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

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

   C#

   ```
   dotnet run
   ```

   Python

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

5. When prompted, speak clearly into the microphone and say "what time is it?". The program should speak, telling you the time.

12.6 Use a different voice

Your speaking clock application uses a default voice, which you can change. The Speech service supports a range of *standard* voices as well as more human-like *neural* voices. You can also create *custom* voices.

Note: For a list of neural and standard voices, see Language and voice support in the Speech service documentation.

1. In the **TellTime** function, under the comment **Configure speech synthesis**, modify the code as follows to specify an alternative voice before creating the **SpeechSynthesizer** client:

   C#

   ```
   // Configure speech synthesis
   speechConfig.SpeechSynthesisVoiceName = "en-GB-George"; // add this
   using SpeechSynthesizer speechSynthesizer = new SpeechSynthesizer(speechConfig);
   ```

   Python

   ```
   # Configure speech synthesis
   speech_config.speech_synthesis_voice_name = 'en-GB-George' # add this
   speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config)
   ```

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

   C#

   ```
   dotnet run
   ```

   Python

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

3. When prompted, speak clearly into the microphone and say ”what time is it?”. The program should speak in the specified voice, telling you the time.

12.7 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**):

   C#

   ```
   // Synthesize spoken output
   string responseSsml = $@"
       <speak version='1.0' xmlns='http://www.w3.org/2001/10/synthesis' xml:lang='en-US'>
           <voice name='en-GB-Susan'>
               {responseText}
               <break strength='weak'/>
               Time to end this lab!
           </voice>
       </speak>";
   SpeechSynthesisResult speak = await speechSynthesizer.SpeakSsmlAsync(responseSsml);
   if (speak.Reason != ResultReason.SynthesizingAudioCompleted)
   {
       Console.WriteLine(speak.Reason);
   }
   ```

   Python

   ```
   # Synthesize spoken output
   responseSsml = " \
       <speak version='1.0' xmlns='http://www.w3.org/2001/10/synthesis' xml:lang='en-US'> \
           <voice name='en-GB-Susan'> \
               {} \
               <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)
```

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

   C#

   ```
   dotnet run
   ```

   Python

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

3. When prompted, speak clearly into the microphone and say "what time is it?". The program should speak in the voice that is specified in the SSML (overriding the voice specified in the SpeechConfig), telling you the time, and then after a pause telling you it's time to end this lab - which it is!

12.8 More information

12.9 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.

12.10 lab: title: 'Translate Speech' module: 'Module 4 - Building Speech-Enabled Applications'

13 Translate Speech

The **Speech** service 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.

Note: This exercise requires that you are using a computer with speakers/headphones. For the best experience, a microphone is also required. Some hosted virtual environments may be able to capture audio from your local microphone, but if this doesn't work (or you don't have a microphone at all), you can use a provided audio file for speech input. Follow the instructions carefully, as you'll need to choose different options depending on whether you are using a microphone or the audio file.

13.1 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/MicrosoftLe` 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**.

13.2 Provision a Cognitive Services resource

If you don't already have on in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*

   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need one of the keys and the location in which the service is provisioned from this page in the next procedure.

13.3 Prepare to use the Speech Translation service

In this exercise, you'll complete a partially implemented client application that uses the Speech SDK to recognize, translate, and synthesize speech.

> **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 **08-speech-translation** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.

2. Right-click the **translator** folder and open an integrated terminal. Then install the Speech SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Microsoft.CognitiveServices.Speech --version 1.14.0
   ```

   Python

   ```
   pip install azure-cognitiveservices-speech==1.14.0
   ```

3. View the contents of the **translator** 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 an authentication **key** for your cognitive services resource, and the **location** where it is deployed. Save your changes.

4. Note that the **translator** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: translator.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 Speech SDK:

   C#

   ```
   // Import namespaces
   using Microsoft.CognitiveServices.Speech;
   using Microsoft.CognitiveServices.Speech.Audio;
   using Microsoft.CognitiveServices.Speech.Translation;
   ```

   Python

   ```
   # Import namespaces
   import azure.cognitiveservices.speech as speech_sdk
   ```

5. In the **Main** function, note that code to load the cognitive services key and region from the configuration file has already been provided. You must use these variables to create a **SpeechTranslationConfig** for your cognitive services resource, which you will use to translate spoken input. Add the following code under the comment **Configure translation**:

   C#

   ```
   // Configure translation
   translationConfig = SpeechTranslationConfig.FromSubscription(cogSvcKey, cogSvcRegion);
   translationConfig.SpeechRecognitionLanguage = "en-US";
   translationConfig.AddTargetLanguage("fr");
   translationConfig.AddTargetLanguage("es");
   ```

```
translationConfig.AddTargetLanguage("hi");
Console.WriteLine("Ready to translate from " + translationConfig.SpeechRecognitionLanguage);
```

Python

```python
# Configure translation
translation_config = speech_sdk.translation.SpeechTranslationConfig(cog_key, cog_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)
```

6. 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**:

   C#

   ```
   // Configure speech
   speechConfig = SpeechConfig.FromSubscription(cogSvcKey, cogSvcRegion);
   ```

   Python

   ```python
   # Configure speech
   speech_config = speech_sdk.SpeechConfig(cog_key, cog_region)
   ```

7. Save your changes and return to the integrated terminal for the **translator** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python translator.py
   ```

8. If you are using C#, you can ignore any warnings about using the **await** operator in asynchronous methods - we'll fix that later. The code should display a message that it is ready to translate from en-US. Press ENTER to end the program.

13.4 Implement speech translation

Now that you have a **SpeechTranslationConfig** for the speech service in your cognitive services resource, you can use the **Speech translation** API to recognize and translate speech.

13.4.1 If you have a working microphone

1. In the **Main** function for your program, note that the code uses the **Translate** function to translate spoken input.

2. 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 using the default system microphone for input.

   C#

   ```
   // Translate speech
   using AudioConfig audioConfig = AudioConfig.FromDefaultMicrophoneInput();
   using TranslationRecognizer translator = new TranslationRecognizer(translationConfig, audioConfig)
   Console.WriteLine("Speak now...");
   TranslationRecognitionResult result = await translator.RecognizeOnceAsync();
   Console.WriteLine($"Translating '{result.Text}'");
   translation = result.Translations[targetLanguage];
   Console.OutputEncoding = Encoding.UTF8;
   Console.WriteLine(translation);
   ```

   Python

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

> **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.

3. Now skip ahead to the **Run the program** section below.

13.4.2 Alternatively, use audio input from a file

1. In the terminal window, enter the following command to install a library that you can use to play the audio file:

   C#

   ```
   dotnet add package System.Windows.Extensions --version 4.6.0
   ```

   Python

   ```
   pip install playsound==1.2.2
   ```

2. In the code file for your program, under the existing namespace imports, add the following code to import the library you just installed:

   C#

   ```
   using System.Media;
   ```

   Python

   ```
   from playsound import playsound
   ```

3. In the **Main** function for your program, 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.

   C#

   ```
   // Translate speech
   string audioFile = "station.wav";
   SoundPlayer wavPlayer = new SoundPlayer(audioFile);
   wavPlayer.Play();
   using AudioConfig audioConfig = AudioConfig.FromWavFileInput(audioFile);
   using TranslationRecognizer translator = new TranslationRecognizer(translationConfig, audioConfig)
   Console.WriteLine("Getting speech from file...");
   TranslationRecognitionResult result = await translator.RecognizeOnceAsync();
   Console.WriteLine($"Translating '{result.Text}'");
   translation = result.Translations[targetLanguage];
   Console.OutputEncoding = Encoding.UTF8;
   Console.WriteLine(translation);
   ```

   Python

   ```
   # Translate speech
   audioFile = 'station.wav'
   playsound(audioFile)
   audio_config = speech_sdk.AudioConfig(filename=audioFile)
   translator = speech_sdk.translation.TranslationRecognizer(translation_config, audio_config)
   print("Getting speech from file...")
   result = translator.recognize_once_async().get()
   print('Translating "{}"'.format(result.text))
   ```

```
translation = result.translations[targetLanguage]
print(translation)
```

> **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.

13.4.3 Run the program

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

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python translator.py
   ```

2. When prompted, enter a valid language code (*fr*, *es*, or *hi*), and then, if using a microphone, speak clearly and say ”where is the station?” or some other phrase you might use when traveling abroad. The program should transcribe your spoken input and translate it to the language you specified (French, Spanish, or Hindi). Repeat this process, trying each language supported by the application. When you’re finished, press ENTER to end the program.

   > **Note**: The TranslationRecognizer gives you around 5 seconds to speak. If it detects no spoken input, it produces a ”No match” result.
   >
   > The translation to Hindi may not always be displayed correctly in the Console window due to character encoding issues.

13.5 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.

1. In the **Translate** function, under the comment **Synthesize translation**, add the following code to use a **SpeechSynthesizer** client to synthesize the translation as speech through the default speaker:

   C#

   ```
   // Synthesize translation
   var voices = new Dictionary<string, string>
                   {
                       ["fr"] = "fr-FR-Julie",
                       ["es"] = "es-ES-Laura",
                       ["hi"] = "hi-IN-Kalpana"
                   };
   speechConfig.SpeechSynthesisVoiceName = voices[targetLanguage];
   using SpeechSynthesizer speechSynthesizer = new SpeechSynthesizer(speechConfig);
   SpeechSynthesisResult speak = await speechSynthesizer.SpeakTextAsync(translation);
   if (speak.Reason != ResultReason.SynthesizingAudioCompleted)
   {
       Console.WriteLine(speak.Reason);
   }
   ```

   Python

   ```
   # Synthesize translation
   voices = {
           "fr": "fr-FR-Julie",
           "es": "es-ES-Laura",
           "hi": "hi-IN-Kalpana"
   }
   speech_config.speech_synthesis_voice_name = voices.get(targetLanguage)
   speech_synthesizer = speech_sdk.SpeechSynthesizer(speech_config)
   speak = speech_synthesizer.speak_text_async(translation).get()
   ```

```
if speak.reason != speech_sdk.ResultReason.SynthesizingAudioCompleted:
    print(speak.reason)
```

2. Save your changes and return to the integrated terminal for the **translator** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python translator.py
   ```

3. When prompted, enter a valid language code (*fr*, *es*, or *hi*), and then speak clearly into the microphone and say a phrase you might use when traveling abroad. The program should transcribe your spoken input and respond with a spoken translation. Repeat this process, trying each language supported by the application. When you're finished, press ENTER to end the program.

   > **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 rather than sent directly to a speaker.*

13.6 More information

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

13.8 lab: title: 'Create a Language Understanding App' module: 'Module 5 - Creating Language Understanding Solutions'

14 Create a Language Understanding App

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.

For example, a language understanding app 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 the language understanding app is to predict the user's intent, and identify any entities to which the intent applies. It is not its job to actually perform the actions required to satisfy the intent. For example, the clock application can use a language app 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.

14.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

14.2 Create Language Understanding resources

To use the Language Understanding service, you need two kinds of resource:

- An *authoring* resource: used to define, train, and test the language understanding app. This must be a **Language Understanding - Authoring** resource in your Azure subscription.
- A *prediction* resource: used to publish your language understanding app and handle requests from client applications that use it. This can be either a **Language Understanding** or **Cognitive Services** resource in your Azure subscription.

  > **Important**: Authoring resources must be created in one of three *regions* (Europe, Australia, or US). Language Understanding apps created in European or Australian authoring resources can only be deployed to prediction resources in Europe or Australia respectively; models created in US authoring resources can be deployed to prediction resources in any Azure location other than Europe and Australia. See the authoring and publishing regions documentation for details about matching authoring and prediction locations.

If you don't already have Language Understanding authoring and prediction resources:

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.

   *Ensure you select **Language Understanding**, not Language Understanding (Azure Cognitive Services)*

   - **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**: *The same as your authoring location*
   - **Prediction pricing tier**: F0
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. If you select **Go to resource**, it will open the *authoring* resource.

14.3 Create a Language Understanding app

Now that you have created an authoring resource, you can use it to create a Language Understanding app.

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.

   > **Note**: If your account is associated with multiple subscriptions in different directories, you may need to switch to the directory containing the subscription where you provisioned your Language Understanding resources.
3. On the **Conversation Apps** page, ensure your subscription and Language Understanding authoring resource are selected. Then create a new app for conversation with the following settings:

   - **Name**: Clock
   - **Culture**: English (*if this option is not available, leave it blank*)
   - **Description**: Natural language clock
   - **Prediction resource**: *Your Language Understanding prediction resource*

   If your **Clock** app isn't opened automatically, open it.

   If a panel with tips for creating an effective Language Understanding app is displayed, close it.

14.4 Create intents

The first thing we'll do in the new app is to define some intents.

1. On the **Intents** page, select **Create** to create a new intent named **GetTime**.
2. In the **GetTime** intent, add the following utterances as example user input:

   what is the time?

   what time is it?
3. After you've added these utterances, go back to the **Intents** page and add another new intent named **GetDay** with the following utterances:

   what is the day today?

   what day is it?
4. After you've added these utterances, go back to the **Intents** page and add another new intent named **GetDate** with the following utterances:

   what is the date today?

   what date is it?
5. After you've added these utterances, go back to the **Intents** page and select the **None** intent. This is provided as a fallback for input that doesn't map to any of the intents you have defined in your language model.
6. Add the following utterances to the **None** intent:

   hello

   goodbye

14.5 Train and test the app

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

1. At the top right of the portal, select **Train** to train the app.
2. When the app is trained, select **Test** to display the Test panel, and then enter the following test utterance:

   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.
3. Try the following test utterance:

   tell me the time

   Again, review the predicted intent and confidence score.
4. Try the following test utterance:

   what's today?

   Hopefully the model predicts the **GetDay** intent.
5. Finally, try this test utterance:

   hi

   This should return the **None** intent.
6. Close the Test panel.

14.6 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.

14.6.1 Add a *machine learned* entity

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

1. On the **Entities** page, select **Create** to create a new entity.
2. In the **Create an entity** dialog box, create a **Machine learned** entity named **Location**.
3. After the **Location** entity has been created, return to the **Intents** page and select the **GetTime** intent.
4. 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:

   what is the current time in New York?
7. When the utterance has been added, select the words ***new york***, and map them to the **Location** entity.

14.6.2 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. On the **Entities** page, select **Create** to create a new entity.
2. In the **Create an entity** dialog box, create a **List** entity named **Weekday**.
3. Add the following **Normalized values** and **synonyms**:

Normalized values	synonyms
sunday	sun
monday	mon
tuesday	tue
wednesday	wed
thursday	thu
friday	fri
saturday	sat

4. After the **Weekday** entity has been created, return to the **Intents** page and select the **GetDate** intent.
5. Enter the following new example utterance:

   what date was it on Saturday?
6. When the utterance has been added, verify that **saturday** has been automatically mapped to the **Weekday** entity. If not, select the word ***saturday***, and in the drop-down list that appears, select **Weekday**.
7. Add another example utterance:

   what date will it be on Friday?
8. When the utterance has been added, ensure **friday** is mapped to the **Weekday** entity.

14.6.3 Add a *Regex* entity

Sometimes, entities have a specific format, such as a serial number, form code, or date. You can define a regular expression (*regex*) that describes an expected format to help your app identify matching entity values.

1. On the **Entities** page, select **Create** to create a new entity.
2. In the **Create an entity** dialog box, create a **Regex** entity named **Date** with the following regex:

   ```
   [0-9]{2}/[0-9]{2}/[0-9]{4}
   ```

Note: This is a simple regex that checks for two digits followed by a "/", another two digits, another "/", and four digits - for example *01/11/2020*. It allows for invalid dates, such as *56/00/9999*; but it's important to remember that the entity regex is used to identify data entry that is *intended* as a date - not to validate date values.

3. After the **Date** entity has been created, return to the **Intents** page and select the **GetDay** intent.
4. Enter the following new example utterance:

   what day was 01/01/1901?
5. When the utterance has been added, verify that **01/01/1901** has been automatically mapped to the **Date** entity. If not, select ***01/01/1901***, and in the drop-down list that appears, select **Date**.
6. Add another example utterance:

   what day will it be on 12/12/2099?
7. When the utterance has been added, ensure **12/12/2099** is mapped to the **Date** entity.

14.6.4 Retrain the app

Now that you've modified ths language model, you need to retrain and retest the app.

1. At the top right of the portal, select **Train** to retrain the app.
2. When the app is trained, select **Test** to display the Test panel, and then enter the following test utterance:

   what's the time in Edinburgh?
3. Review the result that is returned, which should hopefully predict the **GetTime** intent. Then select **Inspect** and in the additional inspection panel that is displayed, examine the **ML entities** section. The model should have predicted that "edinburgh" is an instance of a **Location** entity.
4. Try testing the following utterances:

   what date is it on Friday?

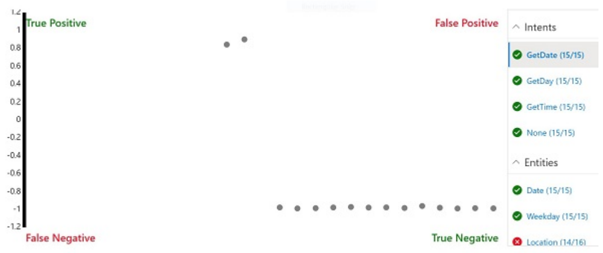
   what's the date on Thu?

   what was the day on 01/01/2020?
5. When you have finished testing, close the inspection panel, but leave the test panel open.

14.7 Perform batch testing

You can use the test pane to test individual utterances interactively, but for more complex language models it is generally more efficient to perform *batch testing*.

1. In Visual Studio Code, open the **batch-test.json** file in the **09-luis-app** folder. This file consists of a JSON document that contains multiple test cases for the clock language model you created.
2. In the Language Understanding portal, in the Test panel, select **Batch testing panel**. Then select **Import** and import the **batch-test.json** file, assigning the name **clock-test**.
3. In the Batch testing panel, run the **clock-test** test.
4. When the test has completed, select **See results**.
5. On the results page, view the confusion matrix that represents the prediction results. It shows true positive, false positive, true negative, and false negative predictions for the intent or entity that is selected in the list on the right.



Note: Each utterance is scored as *positive* or *negative* for each intent - so for example ”what time is it?” should be scored as *positive* for the **GetTime** intent, and *negative* for the **GetDate** intent. The points on the confusion matrix show which utterances were predicted correctly (*true*) and incorrectly (*false*) as *positive* and *negative* for the selected intent.

6. With the **GetDate** intent selected, select any of the points on the confusion matrix to see the details of the prediction - including the utterance and the confidence score for the prediction. Then select the **GetDay**, **GetTime** and **None** intents and view their prediction results. The app should have done well at predicting the intents correctly.

   Note: The user interface may not clear previously selected points.

7. Select the **Location** entity and view the prediction results in the confusion matrix. In particular, note the predictions that were *false negatives* - these were cases where the app failed to detect the specified location in the utterance, indicating that you may need to add more sample utterances to the intents and retrain the model.
8. Close the Batch testing panel.

14.8 Publish the app

In a real project, you’d iteratively refine intents and entities, retrain, and retest until you are satisfied with the predictive performance. Then, you can publish the app for client applications to use.

1. At the top right of the Language Understanding portal, select **Publish**.
2. Select **Production slot**, and publish the app.
3. After publishing is complete, at the top of the Language Understanding portal, select **Manage**.
4. On the **Settings** page, note the **App ID**. Client applications need this to use your app.
5. On the **Azure Resources** page, note the **Primary Key**, **Secondary Key**, and **Endpoint URL** for the prediction resource through which the app can be consumed. Client applications need the endpoint and one of the keys to connect to the prediction resource and be authenticated.
6. In Visual Studio Code, in the **09-luis-app** folder, select the **GetIntent.cmd** batch file and view the code it contains. This command-line script uses cURL to call the Language Understanding REST API for the specified application and prediction endpoint.
7. Replace the placeholder values in the script with the **App ID**, **Endpoint URL**, and either the **Primary Key** or **Secondary Key** for your Language Understanding app; and then save the updated file.
8. Right-click the **09-luis-app** folder and open an integrated terminal. Then enter the following command (be sure to include the quotation marks!):

   ```
   GetIntent "What's the time?"
   ```

9. Review the JSON response returned by your app, which should indicate the top scoring intent predicted for your input (which should be **GetTime**).
10. Try the following command:

    ```
    GetIntent "What's today's date?"
    ```

11. Examine the response and verify that it predicts the **GetDate** intent.
12. Try the following command:

    ```
    GetIntent "What time is it in Sydney?"
    ```

13. Examine the response and verify that it includes a **Location** entity.
14. Try the following commands and examine the responses:

    ```
    GetIntent "What time is it in Glasgow?"

    GetIntent "What's the time in Nairobi?"

    GetIntent "What's the UK time?"
    ```

15. Try a few more variations - the goal is to generate at least some responses that correctly predict the **GetTime** intent, but fail to detect a **Location** entity.

    Keep the terminal open. You will return to it later.

14.9 Apply *active learning*

You can improve a Language Understanding app based on historical utterances submitted to the endpoint. This practice is called *active learning*.

In the previous procedure, you used cURL to submit requests to your app's endpoint. These requests included the option to log the queries, which enables the app to track them for use in active learning.

1. In the Language Understanding portal, Select **Build** and view the **Review endpoint utterances** page. This page lists logged utterances that the service has flagged for review.
2. For any utterances for which the intent and a new location entity (that wasn't included in the original training utterances) are correctly predicted, select to confirm the entity, and then use the icon to add the utterance to the intent as a training example.
3. Find an example of an utterance in which the **GetTime** intent was correctly identified, but a **Location** entity was not identified; and select the location name and map it to the **location** entity. Then use the icon to add the utterance to the intent as a training example.
4. Go to the **Intents** page and open the **GetTime** intent to confirm that the suggested utterances have been added.
5. At the top of the Language Understanding portal, select **Train** to retrain the app.
6. At the top right of the Language Understanding portal, select **Publish** and republish the app to the **Production slot**.
7. Return to the terminal for the **09-luis-app** folder, and use the **GetIntent** command to submit the utterance you added and corrected during active learning.
8. Verify that the result now includes the **Location** entity. Then try another utterance that uses the same phrasing but specifies a different location (for example, *Berlin*).

14.10 Export the app

You can use the Language Understanding portal to develop and test your language app, but in a software development process for DevOps, you should maintain a source controlled definition of the app that can be included in continuous integration and delivery (CI/CD) pipelines. While you *can* use the Language Understanding SDK or REST API in code scripts to create and train the app, a simpler way is to use the portal to create the app, and export it as a *.lu* file that can be imported and retrained in another Language Understanding instance. This approach enables you to make use of the productivity benefits of the portal while maintaining portability and reproducibility for the app.

1. In the Language Understanding portal, select **Manage**.
2. On the **Versions** page, select the current version of the app (there should only be one).
3. In the **Export** drop-down list, select **Export as LU**. Then, when prompted by your browser, save the file in the **09-luis-app** folder.
4. In Visual Studio Code, open the **.lu** file you just exported and downloaded (if you are prompted to search the marketplace for an extension that can read it, dismiss the prompt). Note that the LU format is human-readable, making it an effective way to document the definition of your Language Understanding app in a team development environment.

14.11 More information

14.12 For more information about using the Language Understanding service, see the Language Understanding documentation.

14.13 lab: title: 'Create a Language Understanding Client Application' module: 'Module 5 - Creating Language Understanding Solutions'

15 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).

15.1 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/MicrosoftLe` 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**.

15.2 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.

15.3 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.

15.4 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#

```
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

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

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

3. 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).

4. 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#

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

Python

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

15.5 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#

```
// Create a client for the LU app
var credentials = new Microsoft.Azure.CognitiveServices.Language.LUIS.Runtime.ApiKeyServiceClientCreden
var luClient = new LUISRuntimeClient(credentials) { Endpoint = predictionEndpoint };
```

Python

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

2. 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#

```
// 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,
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

```
# 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.

3. 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#

```
// 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

```
# 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.')
```

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

C#

```
dotnet run
```

Python

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

5. 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.

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

15.6 More information

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

15.8 lab: title: 'Use the Speech and Language Understanding Services' module: 'Module 5 - Creating Language Understanding Solutions'

16 Use the Speech and Language Understanding Services

You can integrate the Speech service with the Language Understanding service to create applications that can intelligently determine user intents from spoken input.

> **Note**: This exercise works best if you have a microphone. Some hosted virtual environments may be able to capture audio from your local microphone, but if this doesn't work (or you don't have a microphone at all), you can use a provided audio file for speech input. Follow the instructions carefully, as you'll need to choose different options depending on whether you are using a microphone or the audio file.

16.1 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/MicrosoftLe` 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**.

16.2 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.

16.3 Prepare a Language Understanding app

If you already have a **Clock** app from a previous exercise, open it in the Language Understanding portal at `https://www.luis.ai`. 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 **11-luis-speech** 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.

16.4 Train and publish the app with *Speech Priming*

1. At the top of the Language Understanding portal, select **Train** to train the app if it has not already been trained.
2. At the top right of the Language Understanding portal, select **Publish**. Then select the **Production slot** and change the settings to enable **Speech Priming** (this will result in better performance for speech recognition).
3. After publishing is complete, at the top of the Language Understanding portal, select **Manage**.
4. On the **Settings** page, note the **App ID**. Client applications need this to use your app.
5. On the **Azure Resources** page, under **Prediction resources**, if no prediction resource is listed, add the prediction resource in your Azure subscription.
6. Note the **Primary Key**, **Secondary Key**, and **Location** (not endpoint!) for the prediction resource. Speech SDK client applications need the location and one of the keys to connect to the prediction resource and be authenticated.

16.5 Configure a client application for Language Understanding

In this exercise, you'll create a client application that accepts spoken input and uses your Language Understanding app to predict the user's intent.

> **Note**: You can choose to use the SDK for either **C#** or **Python** in this exercise. In the steps that follow, perform the actions appropriate for your preferred language.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **11-luis-speech** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. View the contents of the **speaking-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 **Location** (not the full endpoint - for example, *eastus*) and one of the **Keys** for its prediction resource (from the **Manage** page for your app in the Language Understanding portal).

16.6 Install SDK Packages

To use the Speech SDK with the Language Understanding service, you need to install the Speech SDK package for your programming language.

1. In Visual Studio, right-click the **speaking-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#

   ```
   dotnet add package Microsoft.CognitiveServices.Speech --version 1.14.0
   ```

   Python

   ```
   pip install azure-cognitiveservices-speech==1.14.0
   ```

2. Additionally, if your system does not have a working microphone, you will need to use an audio file to provide spoken input for your application. In this case, use the following commands to install an additional package so your program can play the audio file (you can skip this if you intend to use a microphone):

C#

```
dotnet add package System.Windows.Extensions --version 4.6.0
```

Python

```
pip install playsound==1.2.2
```

3. Note that the **speaking-clock-client** folder contains a code file for the client application:
   - **C#**: Program.cs
   - **Python**: speaking-clock-client.py
4. 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 Speech SDK:

C#

```
// Import namespaces
using Microsoft.CognitiveServices.Speech;
using Microsoft.CognitiveServices.Speech.Audio;
using Microsoft.CognitiveServices.Speech.Intent;
```

Python

```
# Import namespaces
import azure.cognitiveservices.speech as speech_sdk
```

5. Additionally, if your system does not have a working microphone, under the existing namespace imports, add the following code to import the library you will use to play an audio file:

C#

```
using System.Media;
```

Python

```
from playsound import playsound
```

16.7 Create an *IntentRecognizer*

The **IntentRecognizer** class provides a client object that you can use to get Language Understanding predictions from spoken input.

1. In the **Main** function, note that code to load the App ID, prediction region, and key from the configuration file has already been provided. Then find the comment **Configure speech service and get intent recognizer**, and add the following code depending on whether you will use a microphone or an audio file for speech input:

16.7.1 If you have a working microphone:

C#

```
// Configure speech service and get intent recognizer
SpeechConfig speechConfig = SpeechConfig.FromSubscription(predictionKey, predictionRegion);
AudioConfig audioConfig = AudioConfig.FromDefaultMicrophoneInput();
IntentRecognizer recognizer = new IntentRecognizer(speechConfig, audioConfig);
```

Python

```
# Configure speech service and get intent recognizer
speech_config = speech_sdk.SpeechConfig(subscription=lu_prediction_key, region=lu_prediction_regio
audio_config = speech_sdk.AudioConfig(use_default_microphone=True)
recognizer = speech_sdk.intent.IntentRecognizer(speech_config, audio_config)
```

16.7.2 If you need to use an audio file:

C#

```
// Configure speech service and get intent recognizer
string audioFile = "time-in-london.wav";
SoundPlayer wavPlayer = new SoundPlayer(audioFile);
wavPlayer.Play();
System.Threading.Thread.Sleep(2000);
SpeechConfig speechConfig = SpeechConfig.FromSubscription(predictionKey, predictionRegion);
AudioConfig audioConfig = AudioConfig.FromWavFileInput(audioFile);
IntentRecognizer recognizer = new IntentRecognizer(speechConfig, audioConfig);
```

Python

```
# Configure speech service and get intent recognizer
audioFile = 'time-in-london.wav'
playsound(audioFile)
speech_config = speech_sdk.SpeechConfig(subscription=lu_prediction_key, region=lu_prediction_regio
audio_config = speech_sdk.AudioConfig(filename=audioFile)
recognizer = speech_sdk.intent.IntentRecognizer(speech_config, audio_config)
```

16.8 Get a predicted intent from spoken input

Now you're ready to implement code that uses the Speech SDK to get a predicted intent from spoken input.

1. In the **Main** function, immediately beneath the code you just added, find the comment **Get the model from the AppID and add the intents we want to use** and add the following code to get your Language Understanding model (based on its App ID) and specify the intents that we want the recognizer to identify.

   C#

   ```
   // Get the model from the AppID and add the intents we want to use
   var model = LanguageUnderstandingModel.FromAppId(luAppId);
   recognizer.AddIntent(model, "GetTime", "time");
   recognizer.AddIntent(model, "GetDate", "date");
   recognizer.AddIntent(model, "GetDay", "day");
   recognizer.AddIntent(model, "None", "none");
   ```

   Note that you can specify a string-based ID for each intent

   Python

   ```
   # Get the model from the AppID and add the intents we want to use
   model = speech_sdk.intent.LanguageUnderstandingModel(app_id=lu_app_id)
   intents = [
       (model, "GetTime"),
       (model, "GetDate"),
       (model, "GetDay"),
       (model, "None")
   ]
   recognizer.add_intents(intents)
   ```

2. Under the comment **Process speech input**, add the following code, which uses the recognizer to asynchronously call the Language Understanding service with spoken input, and retrieve response. If the response includes a predicted intent, the spoken query, predicted intent, and full JSON response are displayed. Otherwise the code handles the response based on the reason returned.

C#

```
// Process speech input
string intent = "";
var result = await recognizer.RecognizeOnceAsync().ConfigureAwait(false);
if (result.Reason == ResultReason.RecognizedIntent)
{
    // Intent was identified
    intent = result.IntentId;
    Console.WriteLine($"Query: {result.Text}");
    Console.WriteLine($"Intent Id: {intent}.");
```

```
    string jsonResponse = result.Properties.GetProperty(PropertyId.LanguageUnderstandingServiceResponse
    Console.WriteLine($"JSON Response:\n{jsonResponse}\n");

    // Get the first entity (if any)

    // Apply the appropriate action

}
else if (result.Reason == ResultReason.RecognizedSpeech)
{
    // Speech was recognized, but no intent was identified.
    intent = result.Text;
    Console.Write($"I don't know what {intent} means.");
}
else if (result.Reason == ResultReason.NoMatch)
{
    // Speech wasn't recognized
    Console.WriteLine($"Sorry. I didn't understand that.");
}
else if (result.Reason == ResultReason.Canceled)
{
    // Something went wrong
    var cancellation = CancellationDetails.FromResult(result);
    Console.WriteLine($"CANCELED: Reason={cancellation.Reason}");
    if (cancellation.Reason == CancellationReason.Error)
    {
        Console.WriteLine($"CANCELED: ErrorCode={cancellation.ErrorCode}");
        Console.WriteLine($"CANCELED: ErrorDetails={cancellation.ErrorDetails}");
    }
}
```

Python

```
# Process speech input
intent = ''
result = recognizer.recognize_once_async().get()
if result.reason == speech_sdk.ResultReason.RecognizedIntent:
    intent = result.intent_id
    print("Query: {}".format(result.text))
    print("Intent: {}".format(intent))
    json_response = json.loads(result.intent_json)
    print("JSON Response:\n{}\n".format(json.dumps(json_response, indent=2)))

    # Get the first entity (if any)

    # Apply the appropriate action

elif result.reason == speech_sdk.ResultReason.RecognizedSpeech:
    # Speech was recognized, but no intent was identified.
    intent = result.text
    print("I don't know what {} means.".format(intent))
elif result.reason == speech_sdk.ResultReason.NoMatch:
    # Speech wasn't recognized
    print("Sorry. I didn't understand that.")
elif result.reason == speech_sdk.ResultReason.Canceled:
    # Something went wrong
    print("Intent recognition canceled: {}".format(result.cancellation_details.reason))
    if result.cancellation_details.reason == speech_sdk.CancellationReason.Error:
        print("Error details: {}".format(result.cancellation_details.error_details))
```

The code you've added so far identifies the *intent*, but some intents can reference *entities*, so you must add code to extract the entity information from the JSON returned by the service.

3. In the code you just added, find the comment **Get the first entity (if any)** and add the following code beneath it:

C#

```
// Get the first entity (if any)
JObject jsonResults = JObject.Parse(jsonResponse);
string entityType = "";
string entityValue = "";
if (jsonResults["entities"].HasValues)
{
    JArray entities = new JArray(jsonResults["entities"][0]);
    entityType = entities[0]["type"].ToString();
    entityValue = entities[0]["entity"].ToString();
    Console.WriteLine(entityType + ": " + entityValue);
}
```

Python

```
# Get the first entity (if any)
entity_type = ''
entity_value = ''
if len(json_response["entities"]) > 0:
    entity_type = json_response["entities"][0]["type"]
    entity_value = json_response["entities"][0]["entity"]
    print(entity_type + ': ' + entity_value)
```

Your code now uses the Language Understanding app to predict an 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.

4. Beneath the code you just added, 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#

```
// Apply the appropriate action
switch (intent)
{
    case "time":
        var location = "local";
        // Check for entities
        if (entityType == "Location")
        {
            location = entityValue;
        }
        // Get the time for the specified location
        var getTimeTask = Task.Run(() => GetTime(location));
        string timeResponse = await getTimeTask;
        Console.WriteLine(timeResponse);
        break;
    case "day":
        var date = DateTime.Today.ToShortDateString();
        // Check for entities
        if (entityType == "Date")
        {
            date = entityValue;
        }
        // Get the day for the specified date
        var getDayTask = Task.Run(() => GetDay(date));
        string dayResponse = await getDayTask;
        Console.WriteLine(dayResponse);
        break;
    case "date":
        var day = DateTime.Today.DayOfWeek.ToString();
        // Check for entities
        if (entityType == "Weekday")
        {
            day = entityValue;
        }

        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("You said " + result.Text.ToLower());
        if (result.Text.ToLower().Replace(".", "") == "stop")
        {
            intent = result.Text;
        }
        else
        {
            Console.WriteLine("Try asking me for the time, the day, or the date.");
        }
        break;
}
```

Python

```python
# Apply the appropriate action
if intent == 'GetTime':
    location = 'local'
    # Check for entities
    if entity_type == 'Location':
        location = entity_value
    # Get the time for the specified location
    print(GetTime(location))

elif intent == 'GetDay':
    date_string = date.today().strftime("%m/%d/%Y")
    # Check for entities
    if entity_type == 'Date':
        date_string = entity_value
    # Get the day for the specified date
    print(GetDay(date_string))

elif intent == 'GetDate':
    day = 'today'
    # Check for entities
    if entity_type == 'Weekday':
        # List entities are lists
        day = entity_value
    # Get the date for the specified day
    print(GetDate(day))

else:
    # Some other intent (for example, "None") was predicted
    print('You said {}'.format(result.text))
    if result.text.lower().replace('.', '') == 'stop':
        intent = result.text
    else:
        print('Try asking me for the time, the day, or the date.')
```

16.9 Run the client application

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

   C#

   ```
   dotnet run
   ```

   Python

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

2. If using a microphone, speak utterances aloud to test the application. For example, try the following (re-running the program each time):

   What's the time?

   What time is it?

   What day is it?

   What is the time in London?

   What's the date?

   What date is Sunday?

   Note: The logic in the application is deliberately simple, and has a number of limitations, but should serve the purpose of testing the ability for the Language Understanding model to predict intents from spoken input using the Speech SDK. You may have trouble recognizing the **GetDay** intent with a specific date entity due to the difficulty in verbalizing a date in *MM/DD/YYYY* format!

16.10 More information

16.11 To learn more about Speech and Language Understanding integration, see the Speech documentation.

16.12 lab: title: 'Create a QnA Solution' module: 'Module 6 - Building a QnA Solution'

17 Create a QnA Solution

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.

QnA Maker is a cognitive service 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.

17.1 Create a QnA Maker resource

To create and host a knowledge base, you need a QnA Maker 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 the **Create a resource** button, search for *QnA*, and create a **QnA Maker** resource with the following settings:
   - **Managed**: Selected
   - **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*
   - **Location**: *Choose any available location*
   - **Pricing tier**: Standard S0
   - **Azure Search location***: *Choose a location in the same global region as your QnA Maker resource.*

- **Azure Search pricing tier**: Free (F) (*If this tier is not available, select Basic (B)*)

  *QnA Maker uses Azure Search to index and query the knowledge base of questions and answers.

3. Select the legal terms checkbox and create the resource.
4. Wait for deployment to complete, and then view the deployment details.

17.2 Create a knowledge base

To create a knowledge base in your QnA Maker resource, you can use the QnA Maker portal. 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 QnA Maker portal at `https://qnamaker.ai` and sign in using the Microsoft account associated with your Azure subscription.
2. At the top of the portal, select **Create a knowledge base**.
3. You have already created a QnA Maker resource, so you can skip step 1. In the **Step 2** section, select the following settings:
   - **Microsoft Azure Directory ID**: The Azure directory containing your subscription.
   - **Azure subscription name**: Your Azure subscription.
   - **Azure QnA Service**: The QnA Maker resource you created previously.
   - **Language**: English (*by default, this option is only available for the first knowledge base you create*).
4. In the **Step 3** section, enter **Learn FAQ** as the name for your 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.
5. In the **Step 4** section:
   - In the **URL** box type `https://docs.microsoft.com/en-us/learn/support/faq` and click **Add URL**.
   - In the **Chit-chat** section, select **Friendly**.
6. In the **Step 5** section, select **Create your KB** and wait for your knowledge base to be created.

17.3 Modify 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 the knowledge base, select **Add QnA pair**.
2. In the **Question** box, type `What is Microsoft certification?`
3. Select **Add alternative phrasing** and type `How can I demonstrate my Microsoft technology skills?`.
4. In the **Answer** box, type `The Microsoft Certified Professional program enables you to validate and prove your skills with Microsoft technologies.`

   In some cases, it makes sense to enable the user to follow up on answer to create a *multi-turn* conversation that enables the user to iteratively refine the question to get to the anseer they need.
5. Under the answer you entered for the certification question, select **Add follow-up prompt**.
6. In the **Follow-up Prompt** dialog box, enter the following settings:
   - **Display text**: `Learn more about certification.`
   - **Link to QnA***: `You can learn more about certification on the [Microsoft certification page](https://docs.microsoft.com/learn/certifications/).`
   - **Context-only**: Selected. *This option ensures that the answer is only ever returned in the context of a follow-up question from the original certification question.*

   *Typing in the **Link to QnA** box searches the existing answers in the knowledge base. When no match is found, it defaults to creating a new QnA pair. Note that the text you type here is in Markdown format.

17.4 Train and test the knowledge base

Now that you have a knowledge base, you can test it in the QnA Maker portal.

1. At the top right of the page, click **Save and train** to train your knowledge base.
2. After training has completed, click ← **Test** to open the test pane.
3. In the test pane, 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 *That makes me happy!* An appropriate chit-chat response should be returned.
6. Enter the message *Tell me about certification.* The answer you created should be returned along with a follow-up prompt button.
7. Select the **Learn more about certification** follow-up button. The follow-up answer with a link to the certification page should be returned.
8. When you're done testing the knowledge base, click → **Test** to close the test pane.

17.5 Publish 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. At the top of the QnA Maker page, click **Publish**. Then in the **Learn FAQ** page, click **Publish**.
2. When publishing is complete, view the sample code provided to use your knowledge base's REST endpoint. There is an example for *Postman* and an example for *Curl*.
3. View the **Curl** tab and copy the example code.
4. Start Visual Studio Code and open a terminal pane.
5. Paste the code you copied into the terminal, and then edit it to replace **<your question>** with **What is a learning path?**.
6. Enter the command and view the JSON response that is returned from your knowledge base.

17.6 Create a bot for the knowledge base

Most commonly, the client applications used to retrieve answers from a knowledge base are bots.

1. On the page containing the publishing confirmation and sample Curl code, select **Create Bot**. This opens the Azure portal in a new browser tab so you can create a Web App Bot in your Azure subscription.

2. In the Azure portal, create a Web App Bot with the following settings (most of these will be pre-populated for you):

   If some values are missing, refresh your browser.

- **Bot handle**: *A unique name for your bot*
- **Subscription**: *Your Azure subscription*
- **Resource group**: *The resource group containing your QnA Maker resource*
- **Location**: *The same location as your QnA Maker service.*
- **Pricing tier**: F0
- **App name**: *Same as the **Bot handle** with *.azurewebsites.net* appended automatically
- **SDK language**: *Choose either C# or Node.js*
- **QnA Auth Key**: *This should automatically be set to the authentication key for your QnA knowledge base*
- **App service plan/location**: *This may be set automatically to a suitable plan and location if one exists. If not, create a new plan*
- **Application Insights**: Off
- **Microsoft App ID and password**: Auto create App ID and password.

3. Wait for your bot to be created (the notification icon at the top right, which looks like a bell, will be animated while you wait). Then in the notification that deployment has completed, click **Go to resource** (or alternatively, on the home page, click **Resource groups**, open the resource group where you created the web app bot, and click it.)
4. In the blade for your bot, view the **Test in Web Chat** page, and wait until the bot displays the message **Hello and welcome!** (it may take a few seconds to initialize).
5. Use the test chat interface to ensure your bot answers questions from your knowledge base as expected. For example, try submitting *What is Microsoft certification?*.

17.7 More information

17.8 To learn more about QnA Maker, see the QnA Maker documentation.

17.9 lab: title: 'Create a Bot with the Bot Framework SDK' module: 'Module 7 - Conversational AI and the Azure Bot Service'

18 Create a Bot with the Bot Framework SDK

Bots are software agents that can participate in conversational dialogs with human users. The Microsoft Bot Framework provides a comprehensive platform for building bots that can be delivered as cloud services through the Azure Bot Service.

In this exercise, you'll use the Microsoft Bot Framework SDK to create and deploy a bot.

18.1 Before you start

Let's start by preparing the environment for bot development.

18.1.1 Update the Bot Framework Emulator

You're going to use the Bot Framework SDK to create your bot, and the Bot Framework Emulator to test it. The Bot Framework Emulator is updated regularly, so let's make sure you have the latest version installed.

> **Note**: Updates may include changes to the user interface that affect the instructions in this exercise.

1. Start the **Bot Framework Emulator**, and if you are prompted to install an update, do so for the currently logged in user. If you are not prompted automatically, use the **Check for update** option on the **Help** menu to check for updates.
2. After installing any available update, close the Bot Framework Emulator until you need it again later.

18.1.2 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.

2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

18.2 Create a bot

You can use the Bot Framework SDK to create a bot based on a template, and then customize the code to meet your specific requirements.

> **Note**: In this exercise, you can choose to use 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 **13-bot-framework** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the folder for your chosen language and open an integrated terminal.
3. In the terminal, run the following commands to install the bot templates and packages you need:

C#

```
dotnet new -i Microsoft.Bot.Framework.CSharp.EchoBot
dotnet new -i Microsoft.Bot.Framework.CSharp.CoreBot
dotnet new -i Microsoft.Bot.Framework.CSharp.EmptyBot
```

Python

```
pip install botbuilder-core
pip install asyncio
pip install aiohttp
pip install cookiecutter==1.7.0
```

4. After the templates and packages have been installed, run the following command to create a bot based on the *EchoBot* template:

C#

```
dotnet new echobot -n TimeBot
```

Python

```
cookiecutter https://github.com/microsoft/botbuilder-python/releases/download/Templates/echo.zip
```

If you're using Python, when prompted by cookiecutter, enter the following details:

- **bot_name**: TimeBot
- **bot_description**: A bot for our times

5. In the terminal pane, enter the following commands to change the current directory to the **TimeBot** folder list the code files that have been generated for your bot:

   ```
   cd TimeBot
   dir
   ```

18.3 Test the bot in the Bot Framework Emulator

You've created a bot based on the *EchoBot* template. Now you can run it locally and test it by using the Bot Framework Emulator (which should be installed on your system).

1. In the terminal pane, ensure that the current directory is the **TimeBot** folder containing your bot code files, and then enter the following command to start your bot running locally.

C#

```
dotnet run
```

Python

```
python app.py
```

When the bot starts, note the endpoint at which it is running is shown. This should be similar to **http://localhost:3978**.

2. Start the Bot Framework Emulator, and open your bot by specifying the endpoint with the **/api/messages** path appended, like this:

   ```
   http://localhost:3978/api/messages
   ```

3. After the conversation is opened in a **Live chat** pane, wait for the message *Hello and welcome!*.

4. Enter a message such as *Hello* and view the response from the bot, which should echo back the message you entered.

5. Close the Bot Framework Emulator and return to Visual Studio Code, then in the terminal window, enter **CTRL+C** to stop the bot.

18.4 Modify the bot code

You've created a bot that echoes the user's input back to them. It's not particularly useful, but serves to illustrate the basic flow of a conversational dialog. A conversation with a bot consists of a sequence of *activities*, in which text, graphics, or user interface *cards* are used to exchange information. The bot begins the conversation with a greeting, which is the result of a *conversation update* activity that is triggered when a user initializes a chat session with the bot. Then the conversation consists of a sequence of further activities in which the user and bot take it in turns to send *messages*.

1. In Visual Studio Code, open the following code file for your bot:

   - **C#**: TimeBot/Bots/EchoBot.cs
   - **Python**: TimeBot/bot.py

Note that the code in this file consists of *activity handler* functions; one for the *Member Added* conversation update activity (when someone joins the chat session) and another for the *Message* activity (when a message is received). The conversation is based on the concept of *turns*, in which each turn represents an interaction in which the bot receives, processes, and responds to an activity. The *turn context* is used to track information about the activity being processed in the current turn.

2. At the top of the code file, add the following namespace import statement:

C#

```
using System;
```

Python

```
from datetime import datetime
```

3. Modify the activity handler function for the *Message* activity to match the following code:

C#

```
protected override async Task OnMessageActivityAsync(ITurnContext<IMessageActivity> turnContext, Cancel
{
    string inputMessage = turnContext.Activity.Text;
    string responseMessage = "Ask me what the time is.";
    if (inputMessage.ToLower().StartsWith("what") && inputMessage.ToLower().Contains("time"))
    {
        var now = DateTime.Now;
        responseMessage = "The time is " + now.Hour.ToString() + ":" + now.Minute.ToString("D2");
    }
    await turnContext.SendActivityAsync(MessageFactory.Text(responseMessage, responseMessage), cancella
}
```

Python

```
async def on_message_activity(self, turn_context: TurnContext):
    input_message = turn_context.activity.text
    response_message = 'Ask me what the time is.'
    if (input_message.lower().startswith('what') and 'time' in input_message.lower()):
        now = datetime.now()
        response_message = 'The time is {}:{:02d}.'.format(now.hour,now.minute)
    await turn_context.send_activity(response_message)
```

4. Save your changes, and then in the terminal pane, ensure that the current directory is the **TimeBot** folder containing your bot code files, and then enter the following command to start your bot running locally.

C#

```
dotnet run
```

Python

```
python app.py
```

As before, when the bot starts, note the endpoint at which it is running is shown.

5. Start the Bot Framework Emulator, and open your bot by specifying the endpoint with the **/api/messages** path appended, like this:

   `http://localhost:3978/api/messages`

6. After the conversation is opened in a **Live chat** pane, wait for the message *Hello and welcome!*.

7. Enter a message such as *Hello* and view the response from the bot, which should be *Ask me what the time is.*

8. Enter *What is the time?* and view the response.

   The bot now responds to the query "What is the time?" by displaying the local time where the bot is running. For any other query, it prompts the user to ask it what the time is. This is a very limited bot, which could be improved through integration with the Language Understanding service and additional custom code, but it serves as a working example of how you can build a solution with the Bot Framework SDK by extending a bot created from a template.

9. Close the Bot Framework Emulator and return to Visual Studio Code, then in the terminal window, enter **CTRL+C** to stop the bot.

18.5 If time permits: Deploy the bot to Azure

Now you're ready to deploy your bot to Azure. Deployment involves multiple steps to prepare the code for deployment and create the necessary Azure resources.

18.5.1 Create or select a resource group

A bot relies on multiple Azure resources, which can be created in a single resource group.

1. Open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.
2. View the **Resource Groups** page to see the resource groups that exist in your subscription.
3. Create a new resource group with a unique name in any available region. (If you are using a "sandbox" subscription that restricts you to an existing resource group, note the resource group name).

18.5.2 Create an Azure application registration

Your bot needs an application registration to enable it to communicate with users and web services.

1. In the terminal window for your **TimeBot** folder, enter the following command to use the Azure command line interface (CLI) to log into Azure. When a browser opens, sign into your Azure subscription.

```
az login
```

2. If you have multiple Azure subscriptions, enter the following command to select the subscription in which you want to deploy the bot.

```
az account set --subscription "<YOUR_SUBSCRIPTION_ID>"
```

3. Enter the following command to create an application registration for **TimeBot** with the password **Super$ecretPassw0rd** (you can use an alternative display name and password if you wish, but make a note of them - you'll need them later).

```
az ad app create --display-name "TimeBot" --password "Super$ecretPassw0rd" --available-to-other-tenants
```

4. When the command completes, a large JSON response is displayed. In this response, find the **appId** value and make a note of it. You will need it in the next procedure.

18.5.3 Create Azure resources

When you use the Bot Framework SDK to create a bot from a template, the Azure Resource Manager templates necessary to create the required Azure resources are provided for you.

1. In the terminal pane for your **TimeBot** folder, enter the following command (on a single line), replacing the PLACEHOLDER values as follows:
   - **YOUR_RESOURCE_GROUP**: The name of your existing resource group.
   - **YOUR_APP_ID**: The **appId** value you noted in the previous procedure.
   - **REGION**: An Azure region code (such as *eastus*).
   - **All other placeholders**: Unique values that will be used to name the new resources. The resource IDs you specify must be globally unique strings netween 4 and 42 characters long. Make a note of the value you use for the **BotId** and **newWebAppName** parameters - you will need them later.

```
az deployment group create --resource-group "YOUR_RESOURCE_GROUP" --template-file "deploymenttemplates/
```

2. Wait for the command to complete. If it is successful, a JSON response will be displayed.

   If an error occurs, it may be caused by a typo in the command or a unique naming conflict with an existing resource. Correct the issue and try again. You may need to use the Azure portal to delete any resources that were created before the failure occurred.

3. After the command has completed, view your resource group in the Azure portal to see the resources that have been created.

18.5.4 Prepare the bot code for deployment

Now that you have the required Azure resources in place, you can prepare your code for deployment to them.

1. In Visual Studio Code, in the terminal pane for your **TimeBot** folder, enter the following command to prepare your code's dependencies for deployment.

C#

```
az bot prepare-deploy --lang Csharp --code-dir "." --proj-file-path "TimeBot.csproj"
```

Python

```
rmdir /S /Q  __pycache__
notepad requirements.txt
```

- The second command will open the requirements.txt file for your Python environment in Notepad - modify it to match the following, save the changes, and close Notepad.

```
botbuilder-core==4.11.0
aiohttp
```

18.5.5 Create a zip archive for deployment

To deploy the bot files, you will package them in a .zip archive. This must> be created from the files and folders in the root folder for your bot (do not zip the root folder itself - zip its contents!).

1. In Visual Studio Code, in the **Explorer** pane, right-click any of the files or folders in your **TimeBot** folder, and select **Reveal in File Explorer**.
2. In the File Explorer window, select all of the files in the **TimeBot** folder. Then right-click any of the selected files and select **Send to** > **Compressed (zipped) folder**.
3. Rename the resulting zipped file in your **TimeBot** folder to **TimeBot.zip**.

18.5.6 Deploy and test the bot

Now that your code is prepared, you can deploy it.

1. In Visual Studio Code, in the terminal pane for your **TimeBot** folder, enter the following command (on a single line) to deploy your packaged code files, replacing the PLACEHOLDER values as follows:
   - **YOUR_RESOURCE_GROUP**: The name of your existing resource group.
   - **YOUR_WEB_APP_NAME**: The unique name you specified for the **newWebAppName** parameter when creating Azure resources.

```
az webapp deployment source config-zip --resource-group "YOUR_RESOURCE_GROUP" --name "YOUR_WEB_APP_NAME
```

2. In the Azure portal, in the resource group containing your resources, open the **Bot Channels Registration** resource (which will have the name you assigned to the **BotId** parameter when creating Azure resources).
3. In the **Bot management** section, select **Test in Web Chat**. Then wait for your bot to initialize.
4. Enter a message such as *Hello* and view the response from the bot, which should be *Ask me what the time is.*
5. Enter *What is the time?* and view the response.

18.6 Use the Web Chat channel in a web page

One of the key benefits of the Azure Bot Service is the ability to deliver your bot through multiple *channels.*

1. In the Azure portal, on the blade for your Bot, view the **Channels** your bot is connected to.
2. Note that the **Web Chat** channel has been added automatically, and that other channels for common communication platforms are available.
3. Next to the **Web Chat** channel, click **Edit**. This opens a page with the settings you need to embed your bot in a web page. To embed your bot, you need the HTML embed code provided as well as one of the secret keys generated for your bot.
4. Copy the **Embed code**.
5. In Visual Studio Code, expand the **13-bot-framework/web-client** folder and select the **default.html** file it contains.
6. In the HTML code, paste the embed code you copied directly beneath the comment **add the iframe for the bot here**

7. Back in the Azure portal, select **Show** for one of your secret keys (it doesn't matter which one), and copy it. Then return to Visual Studio Code and paste it in the HTML embed code you added previously, replacing **YOUR_SECRET_HERE**.
8. In Visual Studio Code, in the **Explorer** pane, right-click **default.html** and select **Reveal in File Explorer**.
9. In the File Explorer window, open **default.html** in Microsoft Edge.
10. In the web page that opens, test the bot by entering *Hello*. Note that it won't initialize until you submit a message, so the greeting message will be followed immediately by a prompt to ask what the time is.
11. Test the bot by submitting *What is the time?*.

18.7 More information

18.8 To learn more about the Bot Framework, view the Bot Framework documentation.

18.9 lab: title: 'Create a Bot with Bot Framework Composer' module: 'Module 7 - Conversational AI and the Azure Bot Service'

19 Create a Bot with Bot Framework Composer

Bot Framework Composer is a graphical designer that lets you quickly and easily build sophisticated conversational bots without writing code. The composer is an open-source tool that presents a visual canvas for building bots.

19.1 Prepare to develop a bot

Let's start by preparing the services and tools you need to develop a bot.

19.1.1 Get an OpenWeather API key

In this exercise, you will create a bot that uses the OpenWeather service to retrieve weather conditions for the city entered by the user. You will require an API key for the service to work.

1. In a web browser, go to the OpenWeather site at `https://openweathermap.org/price`.
2. Request a free API key, and create an OpenWeather account (if you do not already have one).
3. After signing up, view the **API keys** page to see your API key.

19.1.2 Update Bot Framework Composer

You're going to use the Bot Framework Composer to create your bot. This tools is updated regularly, so let's make sure you have the latest version installed.

> **Note**: Updates may include changes to the user interface that affect the instructions in this exercise.

1. Start the **Bot Framework Composer**, and if you are not automatically prompted to install an update, use the **Check for updates** option on the **Help** menu to check for updates.
2. If an update is available, choose the option to install it when the application is closed. Then close the Bot Framework Composer and install the update for the currently logged in user, restarting the Bot Framework Composer after the installation is complete. Installation may take a few minutes.
3. Ensure that the version of Bot Framework Composer is **1.4.0** or later.

19.2 Create a bot

Now you're ready to use the Bot Framework Composer to create a bot.

19.2.1 Create a bot and customize the "welcome" dialog flow

1. Start the Bot Framework Composer if it's not already open.

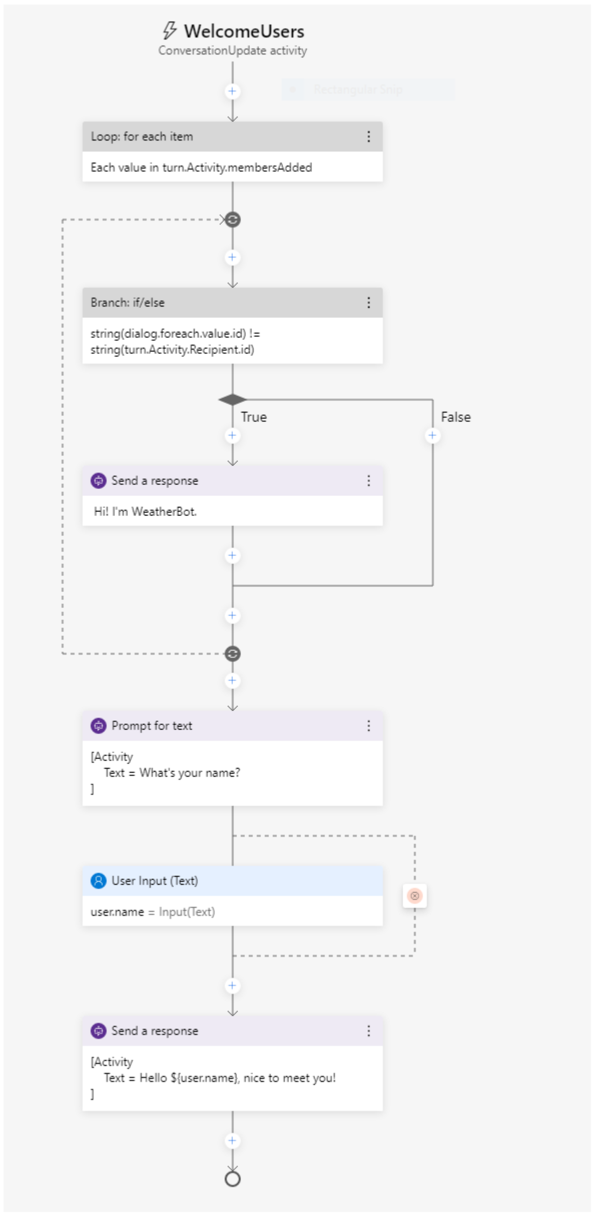
2. On the **Home** screen, select **New**. Then create a new bot from scratch; naming it **WeatherBot** and saving it in a local folder.

3. In the navigation pane on the left, select **Greeting** to open the authoring canvas and show the *ConversationUpdate* activity that is called when a user initially joins a conversation with the bot. The activity consists of a flow of actions.

4. In the properties pane on the right, edit the title of **Greeting** by selecting the word **Greeting** at the top of the properties pane on the right and changing it to **WelcomeUsers**.
5. In the authoring canvas, select the **Send a response** action. Then, in the properties pane, change the default text from - *${WelcomeUser()}* to `- Hi! I'm WeatherBot.` (including the preceding "-" dash).
6. In the authoring canvas, select the final + symbol (just above the circle that marks the end of the dialog flow), and add a new **Ask a question** action for a **Text** response.

   The new action creates two nodes in the dialog flow. The first node defines a prompt for the bot to ask the user a question, and the second node represents the response that will be received from the user. In the properties pane, these nodes have corresponding **Bot asks** and **User input** tabs.
7. In the properties pane, on the **Bot Asks** tab, set the **Prompt for text** value to `What's your name?`. Then, on the **User Input** tab, set the **Property** value to `user.name` to define a variable that you can access later in the bot conversation.
8. Back in the authoring canvas, select the + symbol under the **User Input(Text)** action you just added, and add a **Send a response** action.
9. Select the newly added **Send a response** action and in the properties pane, set the text value to `Hello ${user.name}, nice to meet you!`.

   The completed activity flow should look like this:



19.2.2 Test the bot

Your basic bot is complete so now let's test it.

1. Select **Start Bot** in the upper right-hand corner of Composer, and wait while your bot is compiled and started. This may take several minutes.
    - If a Windows Firewall message is displayed, enable access for all networks.
2. In the **Local bot runtime manager** pane, select **Open Web Chat**.
3. In the **WeatherBot** web chat pane, after a short pause, you will see the welcome message and the prompt to enter your first name. Enter you first name and press **Enter**.
4. The bot should respond with the **Hello *your_name*, nice to meet you!**.
5. Close the web chat panel.
6. At the top right of Composer, next to **Restart bot**, click = to open the **Local bot runtime manager** pane, and use the icon to stop the bot.

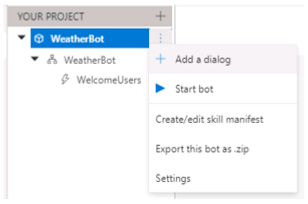
19.3 Add a dialog to get the weather

Now that you have a working bot, you can expand its capabilities by adding dialogs for specific interactions. In this case, you'll add a dialog that is triggered when the user mentions "weather".

19.3.1 Add a dialog

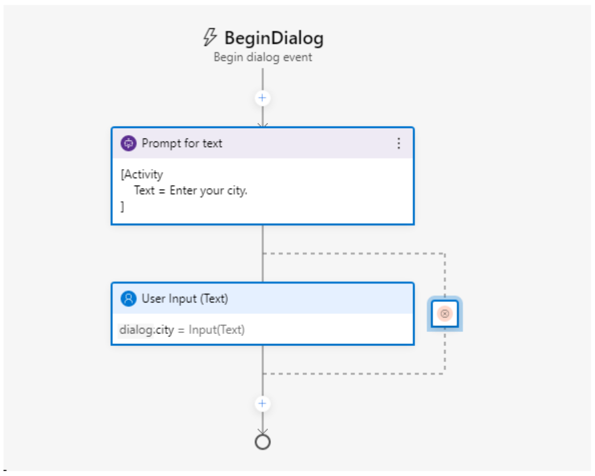
First, you need to define a dialog flow that will be used to handle questions about the weather.

1. In Composer, in the navigation pane, hold the mouse over the top level node (**WeatherBot**) and in the **...** menu, select + **Add a dialog**, as shown here:

   Then create a new dialog named **GetWeather** with the description **Get the current weather condition for the provided zip code**.
2. In the navigation pane, select the **BeginDialog** node for the new **GetWeather** dialog. Then on the authoring canvas, use the + symbol to add a **Ask a question** action for a **Text** response.
3. In the properties pane, on the **Bot asks** tab, set the **Prompt for text** to `Enter your city`.
4. On the **User Input** tab, set the **Property** field to `dialog.city`, and set the **Output format** field to the expression `=trim(this.value)` to remove any superfluous spaces around the user-provided value.

   The activity flow so far should look like this:

   ```
   [Activity
     Text = Enter your city.
   ]
   ```

   ```
   dialog.city = Input(Text)
   ```

So far, the dialog asks the user to enter a city. Now you must implement the logic to retrieve the weather information for the city that was entered.

5. On the authoring canvas, directly under the **user Input** action for the city entry, select the + symbol to add a new action.
6. From the list of actions, select **Access external resources** and then **Send an HTTP request**.
7. Set the properties for the **HTTP request** as follows, replacing **YOUR_API_KEY** with your OpenWeather API key:

   - **HTTP method**: GET
   - **Url**: `http://api.openweathermap.org/data/2.5/weather?units=metric&q=${dialog.city}&appid=YOUR_AP`
   - **Result property**: `dialog.api_response`

   The result can include any of the following four properties from the HTTP response:

   - **statusCode**. Accessed via **dialog.api_response.statusCode**.
   - **reasonPhrase**. Accessed via **dialog.api_response.reasonPhrase**.
   - **content**. Accessed via **dialog.api_response.content**.
   - **headers**. Accessed via **dialog.api_response.headers**.

   Additionally, if the response type is JSON, it will be a deserialized object available via **dialog.api_response.content** property. For detailed information about the OpenWeather API and the response it returns, see the OpenWeather API documentation.

   Now you need to add logic to the dialog flow that handles the response, which might indicate success or failure of the HTTP request.

8. On the authoring canvas, under the **Send HTTP Request** action you created, add a **Create a condition** > **Branch: if/else** action. This action defines a branch in the dialog flow with **True** and **False** paths.
9. In the **Properties** of the branch action, set the **Condition** field to write the following expression:

   `=dialog.api_response.statusCode == 200`

10. If the call was successful, you need to store the response in a variable. On the authoring canvas, in the **True** branch, add a **Manage properties** > **Set properties** action. Then in the properties pane, add the following property assignments:

Property	Value
`dialog.weather`	`=dialog.api_response.content.weather[0].description`
`dialog.temp`	`=round(dialog.api_response.content.main.temp)`
`dialog.icon`	`=dialog.api_response.content.weather[0].icon`

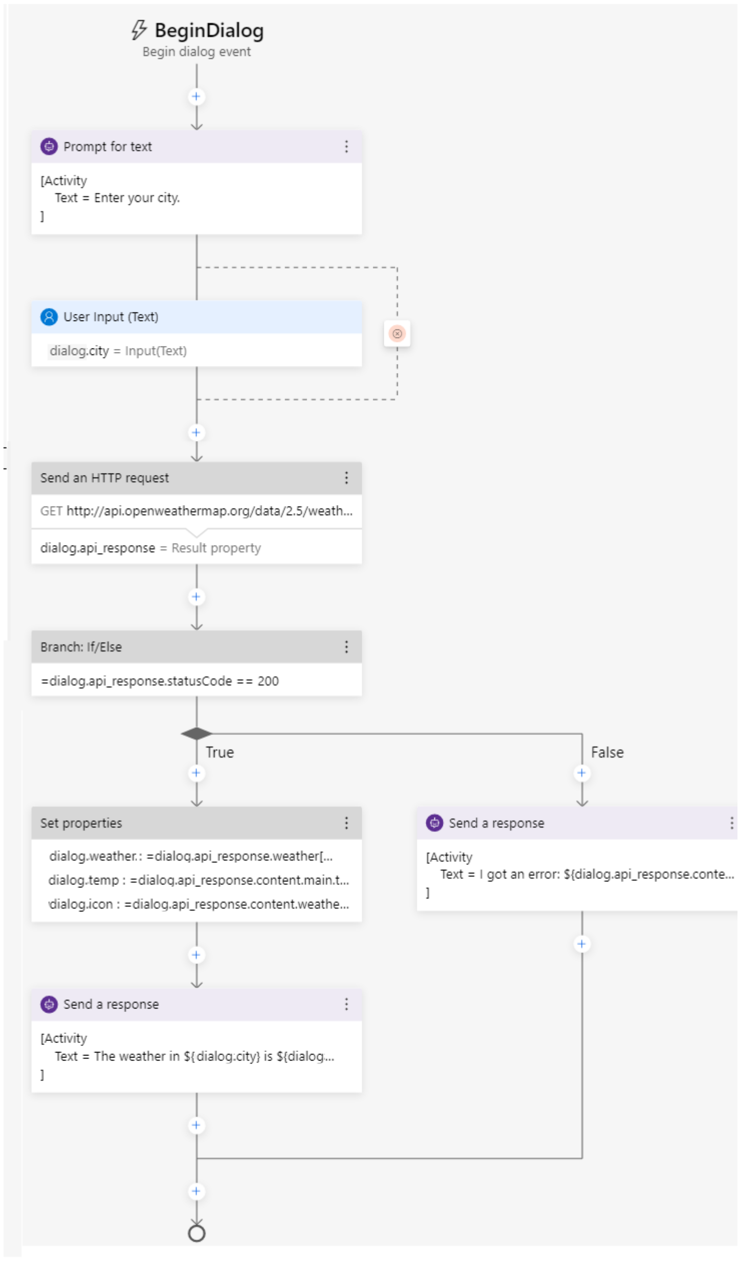
11. Still in the **True** branch, add a **Send a response** action under the **Set a property** action and set its text to:

   `The weather in ${dialog.city} is ${dialog.weather} and the temperature is ${dialog.temp}&deg;.`

   Note: *This message uses the* ***dialog.city***, ***dialog.weather***, *and* ***dialog.temp*** *properties you set in the previous actions. Later, you'll also use the* ***dialog.icon*** *property.*

12. You also need to account for a response from the weather service that is not 200, so in the **False** branch, add a **Send a response** action and set its text to `I got an error: ${dialog.api_response.content.message}.`

   The dialog flow should now look like this:


BeginDialog
Begin dialog event

Prompt for text
```
[Activity
   Text = Enter your city.
]
```

User Input (Text)
```
dialog.city = Input(Text)
```

Send an HTTP request
```
GET http://api.openweathermap.org/data/2.5/weath...
dialog.api_response = Result property
```

Branch: If/Else
```
=dialog.api_response.statusCode == 200
```

True

Set properties
```
dialog.weather.: =dialog.api_response.weather[...
dialog.temp : =dialog.api_response.content.main.t...
dialog.icon : =dialog.api_response.content.weathe...
```

Send a response
```
[Activity
   Text = The weather in ${dialog.city} is ${dialog...
]
```

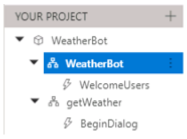
False

Send a response
```
[Activity
   Text = I got an error: ${dialog.api_response.conte...
]
```

19.3.2 Add a trigger for the dialog

Now you need some way for the new dialog to be initiated from the existing welcome dialog.

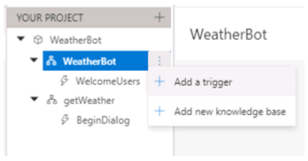
1. In the navigation pane, select the **WeatherBot** dialog that contains **WelcomeUsers** (this is under the top-level bot node of the same name).



2. In the properties pane for the selected **WeatherBot** dialog, in the **Language Understanding** section, set the **Recognizer type** to **Regular expression recognizer**.

   The default recognizer type uses the Language Understanding service to product the user's intent using a natural language understanding model. We're using a regular expression recognizer to simplify this exercise. In a real, application, you should consider using Language Understanding to allow for more sophisticated intent recognition.

3. In the **...** menu for the **WeatherBot** dialog, select **Add a Trigger**.



Then create a trigger with the following settings:

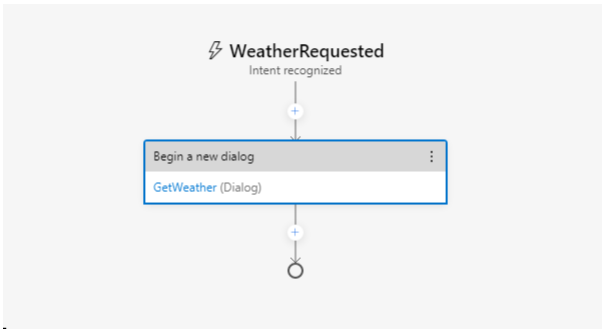
- **What is the type of this trigger?**: Intent recognized
- **What is the name of this trigger (RegEx)**: `WeatherRequested`
- **Please input regex pattern**: `weather`

  The text entered in the regex pattern text box is a simple regular expression pattern that will cause the bot to look for the word *weather* in any incoming message. If "weather" is present, the message becomes a **recognized intent** and the trigger is initiated.

4. Now that the trigger is created, you need to configure an action for it. In the authoring canvas for the trigger, select the + symbol under your new **WeatherRequested** trigger node. Then in the list of actions, select **Dialog Management** and select **Begin a new dialog**.

5. With the **Begin a new dialog** action selected, in the properties pane, select the **GetWeather** dialog from the **Dialog name** drop-down list to start the **GetWeather** dialog you defined earlier when the **WeatherRequested** trigger is recognized.

   The **WeatherRequested** activity flow should look like this:



6. Restart the bot and open the web chat pane.Then restart the conversation, and after entering your name, enter `What is the weather like?`. Then, when prompted, enter a city, such as `Seattle`. The bot will contact the service and should respond with a small weather report statement.

7. When you have finished testing, close the web chat pane and stop the bot.

19.4 Handle interruptions

A well designed bot should allow users to change the flow of the conversation, for example by canceling a request.

1. In the Bot Composer, in the navigation pane, use the **...** menu for the **WeatherBot** dialog to add a new trigger (in addition to the existing **WelcomeUsers** and **WeatherRequested** triggers). The new trigger should have the following settings:

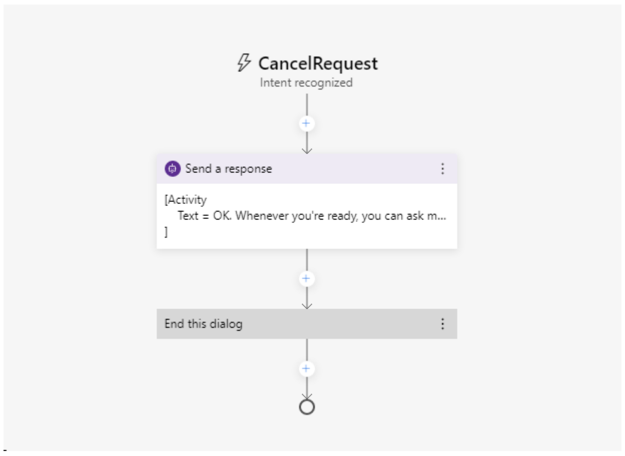
   - **What is the type of this trigger?**: Intent recognized
   - **What is the name of this trigger (RegEx)**: `CancelRequest`
   - **Please input regex pattern**: `cancel`

     The text entered in the regex pattern text box is a simple regular expression pattern that will cause the bot to look for the word *cancel* in any incoming message.

2. In the authoring canvas for the trigger, add a **Send a response** action, and set its **Language Generation** property to `OK. Whenever you're ready, you can ask me about the weather.`

3. Under the **Send a response** action, add a new action to and the dialog by selecting **Dialog management** and **End this dialog**.

The **CancelRequest** dialog flow should look like this:



```
[Activity
    Text = OK. Whenever you're ready, you can ask m...
]
```

Now that you have a trigger to respond to a user's request to cancel, you must allow interruptions to dialog flows where the user might want to make such a request - such as when prompted for a zip code after asking for weather information.

4. In the navigation pane, select **BeginDialog** under the **GetWeather** dialog.
5. Select the **Prompt for text** action that asks the user to enter their city.
6. In the properties for the action, on the **Other** tab, expand **Prompt Configurations** and set the **Allow Interruptions** property to **true**.
7. Restart the bot and open the web chat pane. Restart the conversation, and and after entering your name, enter `What is the weather like?`. Then, when prompted, enter `cancel`, and confirm that the request is canceled.
8. After canceling the request, enter `What's the weather like?` and note that the appropriate trigger starts a new instance of the **GetWeather** dialog, prompting you once again to enter a city.
9. When you have finished testing, close the web chat pane and stop the bot.

19.5 Enhance the user experience

The interactions with the weather bot so far has been through text. Users enter text for their intentions and the bot responds with text. While text is often a suitable way to communicate, you can enhance the experience through other forms of user interface element. For example, you can use buttons to initiate recommended actions, or display a *card* to present information visually.

19.5.1 Add a button

1. In the Bot Framework Composer, in the navigation pane, under the **GetWeather** action, select **BeginDialog**.
2. In the authoring canvas, select the **Prompt for text** action that contains the prompt for the city.
3. In the properties pane, select **Switch to code editor**, and replace the existing code with the following code.

```
[Activity
    Text = What is your city?
    SuggestedActions = Cancel
]
```

This activity will prompt the user for their city as before, but also display a **Cancel** button.

19.5.2 Add a card

1. In the **GetWeather** dialog, in the **True** path after checking the response from the HTTP weather service, select the **Send a response** action that displays the weather report.
2. In the properties pane, select **Switch to code editor** and replace the existing code with the following code.

```
[ThumbnailCard
    title = Weather for ${dialog.city}
    text = ${dialog.weather} (${dialog.temp}&deg;)
    image = http://openweathermap.org/img/w/${dialog.icon}.png
]
```

This template will use the same variables as before for the weather condition but also adds a title to the card that will be displayed, along with an image for the weather condition.

19.5.3 Test the new user interface

1. Restart the bot and open the web chat pane. Restart the conversation, and after entering your name, enter ”What is the weather like?”. Then, when prompted, click the **Cancel** button to cancel the request.
2. After canceling, enter `Tell me about the weather` and when prompted, enter a city, such as `London`. The bot will contact the service and should respond with a card indicating the weather conditions.
3. When you have finished testing, close the emulator and stop the bot.

19.6 More information

19.7 To learn more about Bot Framework Composer, view the Bot Framework Composer documentation.

19.8 lab: title: ’Analyze Images with Computer Vision’ module: ’Module 8 - Getting Started with Computer Vision’

20 Analyze Images with Computer Vision

Computer vision is an artificial intelligence capability that enables software systems to interpret visual input by analyzing images. In Microsoft Azure, the **Computer Vision** cognitive service provides pre-built models for common computer vision tasks, including analysis of images to suggest captions and tags, detection of common objects, landmarks, celebrities, brands, and the presence of adult content. You can also use the Computer Vision service to analyze image color and formats, and to generate ”smart-cropped” thumbnail images.

20.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you’re working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

20.2 Provision a Cognitive Services resource

If you don’t already have one in your subscription, you’ll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.

5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

20.3 Prepare to use the Computer Vision SDK

In this exercise, you'll complete a partially implemented client application that uses the Computer Vision SDK to analyze images.

> **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 **15-computer-vision** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **image-analysis** folder and open an integrated terminal. Then install the Computer Vision SDK package by running the appropriate command for your language preference:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.ComputerVision --version 6.0.0
```

Python

```
pip install azure-cognitiveservices-vision-computervision==0.7.0
```

3. View the contents of the **image-analysis** folder, and note that it contains a file for configuration settings:

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

    Open the configuration file and update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.

4. Note that the **image-analysis** folder contains a code file for the client application:

    - **C#**: Program.cs
    - **Python**: image-analysis.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 Computer Vision SDK:

C#

```
// import namespaces
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
```

Python

```
# import namespaces
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from azure.cognitiveservices.vision.computervision.models import VisualFeatureTypes
from msrest.authentication import CognitiveServicesCredentials
```

20.4 View the images you will analyze

In this exercise, you will use the Computer Vision service to analyze multiple images.

1. In Visual Studio Code, expand the **image-analysis** folder and the **images** folder it contains.
2. Select each of the image files in turn to view then in Visual Studio Code.

20.5 Analyze an image to suggest a caption

Now you're ready to use the SDK to call the Computer Vision service and analyze an image.

1. In the code file for your client application (**Program.cs** or **image-analysis.py**), in the **Main** function, note that the code to load the configuration settings has been provided. Then find the comment **Authenticate Computer Vision client**. Then, under this comment, add the following language-specific code to create and authenticate a Computer Vision client object:

C#

```
// Authenticate Computer Vision client
ApiKeyServiceClientCredentials credentials = new ApiKeyServiceClientCredentials(cogSvcKey);
cvClient = new ComputerVisionClient(credentials)
{
    Endpoint = cogSvcEndpoint
};
```

Python

```
# Authenticate Computer Vision client
credential = CognitiveServicesCredentials(cog_key)
cv_client = ComputerVisionClient(cog_endpoint, credential)
```

2. In the **Main** function, under the code you just added, note that the code specifies the path to an image file and then passes the image path to two other functions (**AnalyzeImage** and **GetThumbnail**). These functions are not yet fully implemented.

3. In the **AnalyzeImage** function, under the comment **Specify features to be retrieved**, add the following code:

C#

```
// Specify features to be retrieved
List<VisualFeatureTypes?> features = new List<VisualFeatureTypes?>()
{
    VisualFeatureTypes.Description,
    VisualFeatureTypes.Tags,
    VisualFeatureTypes.Categories,
    VisualFeatureTypes.Brands,
    VisualFeatureTypes.Objects,
    VisualFeatureTypes.Adult
};
```

Python

```
# Specify features to be retrieved
features = [VisualFeatureTypes.description,
            VisualFeatureTypes.tags,
            VisualFeatureTypes.categories,
            VisualFeatureTypes.brands,
            VisualFeatureTypes.objects,
            VisualFeatureTypes.adult]
```

4. In the **AnalyzeImage** function, under the comment **Get image analysis**, add the following code (including the comments indicating where you will add more code later.):

C#

```
// Get image analysis
using (var imageData = File.OpenRead(imageFile))
{
    var analysis = await cvClient.AnalyzeImageInStreamAsync(imageData, features);

    // get image captions
    foreach (var caption in analysis.Description.Captions)
    {
        Console.WriteLine($"Description: {caption.Text} (confidence: {caption.Confidence.ToString("P")}
    }

    // Get image tags


    // Get image categories


    // Get brands in the image
    // Get objects in the image


    // Get moderation ratings


}
```

Python

```
# Get image analysis
with open(image_file, mode="rb") as image_data:
    analysis = cv_client.analyze_image_in_stream(image_data , features)

# Get image description
for caption in analysis.description.captions:
    print("Description: '{}' (confidence: {:.2f}%)".format(caption.text, caption.confidence * 100))

# Get image tags


# Get image categories


# Get brands in the image


# Get objects in the image


# Get moderation ratings
```

5. Save your changes and return to the integrated terminal for the **image-analysis** folder, and enter the following command to run the program with the argument **images/street.jpg**:

C#

```
dotnet run images/street.jpg
```

Python

```
python image-analysis.py images/street.jpg
```

6. Observe the output, which should include a suggested caption for the **street.jpg** image.
7. Run the program again, this time with the argument **images/building.jpg** to see the caption that gets generated for the **building.jpg** image.
8. Repeat the previous step to generate a caption for the **images/person.jpg** file.

20.6 Get suggested tags for an image

It can sometimes be useful to identify relevant *tags* that provide clues about the contents of an image.

1. In the **AnalyzeImage** function, under the comment **Get image tags**, add the following code:

C#

```
// Get image tags
if (analysis.Tags.Count > 0)
{
    Console.WriteLine("Tags:");
    foreach (var tag in analysis.Tags)
    {
        Console.WriteLine($" -{tag.Name} (confidence: {tag.Confidence.ToString("P")})");
    }
}
```

Python

```python
# Get image tags
if (len(analysis.tags) > 0):
    print("Tags: ")
    for tag in analysis.tags:
        print(" -'{}' (confidence: {:.2f}%)".format(tag.name, tag.confidence * 100))
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function and observing that in addition to the image caption, a list of suggested tags is displayed.

20.7 Get image categories

The Computer Vision service can suggest *categories* for images, and within each category it can identify well-known landmarks or celebrities.

1. In the **AnalyzeImage** function, under the comment **Get image categories (including celebrities and landmarks)**, add the following code:

C#

```csharp
// Get image categories (including celebrities and landmarks)
List<LandmarksModel> landmarks = new List<LandmarksModel> {};
List<CelebritiesModel> celebrities = new List<CelebritiesModel> {};
Console.WriteLine("Categories:");
foreach (var category in analysis.Categories)
{
    // Print the category
    Console.WriteLine($" -{category.Name} (confidence: {category.Score.ToString("P")})");

    // Get landmarks in this category
    if (category.Detail?.Landmarks != null)
    {
        foreach (LandmarksModel landmark in category.Detail.Landmarks)
        {
            if (!landmarks.Any(item => item.Name == landmark.Name))
            {
                landmarks.Add(landmark);
            }
        }
    }

    // Get celebrities in this category
    if (category.Detail?.Celebrities != null)
    {
        foreach (CelebritiesModel celebrity in category.Detail.Celebrities)
        {
            if (!celebrities.Any(item => item.Name == celebrity.Name))
            {
                celebrities.Add(celebrity);
            }
        }
    }
}

// If there were landmarks, list them
if (landmarks.Count > 0)
{
    Console.WriteLine("Landmarks:");
    foreach(LandmarksModel landmark in landmarks)
    {
        Console.WriteLine($" -{landmark.Name} (confidence: {landmark.Confidence.ToString("P")})");
    }
}

// If there were celebrities, list them
if (celebrities.Count > 0)
{
    Console.WriteLine("Celebrities:");
    foreach(CelebritiesModel celebrity in celebrities)
    {
        Console.WriteLine($" -{celebrity.Name} (confidence: {celebrity.Confidence.ToString("P")})");
    }
}
```

Python

```
# Get image categories (including celebrities and landmarks)
if (len(analysis.categories) > 0):
    print("Categories:")
    landmarks = []
    celebrities = []
    for category in analysis.categories:
        # Print the category
        print(" -'{}' (confidence: {:.2f}%)".format(category.name, category.score * 100))
        if category.detail:
            # Get landmarks in this category
            if category.detail.landmarks:
                for landmark in category.detail.landmarks:
                    if landmark not in landmarks:
                        landmarks.append(landmark)

            # Get celebrities in this category
            if category.detail.celebrities:
                for celebrity in category.detail.celebrities:
                    if celebrity not in celebrities:
                        celebrities.append(celebrity)

    # If there were landmarks, list them
    if len(landmarks) > 0:
        print("Landmarks:")
        for landmark in landmarks:
            print(" -'{}' (confidence: {:.2f}%)".format(landmark.name, landmark.confidence * 100))

    # If there were celebrities, list them
    if len(celebrities) > 0:
        print("Celebrities:")
        for celebrity in celebrities:
            print(" -'{}' (confidence: {:.2f}%)".format(celebrity.name, celebrity.confidence * 100))
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function and observing that in addition to the image caption and tags, a list of suggested categories is displayed along with any recognized landmarks or celebrities (in particular in the **building.jpg** and **person.jpg** images).

20.8 Get brands in an image

Some brands are visually recognizable from logo's, even when the name of the brand is not displayed. The Computer Vision service is trained to identify thousands of well-known brands.

1. In the **AnalyzeImage** function, under the comment **Get brands in the image**, add the following code:

C#

```
// Get brands in the image
if (analysis.Brands.Count > 0)
{
    Console.WriteLine("Brands:");
    foreach (var brand in analysis.Brands)
    {
        Console.WriteLine($" -{brand.Name} (confidence: {brand.Confidence.ToString("P")})");
    }
}
```

Python

```
# Get brands in the image
if (len(analysis.brands) > 0):
    print("Brands: ")
    for brand in analysis.brands:
        print(" -'{}' (confidence: {:.2f}%)".format(brand.name, brand.confidence * 100))
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function and observing any brands that are identified (specifically, in the **person.jpg** image).

20.9 Detect and locate objects in an image

Object detection is a specific form of computer vision in which individual objects within an image are identified and their location indicated by a bounding box..

1. In the **AnalyzeImage** function, under the comment **Get objects in the image**, add the following code:

C#

```
// Get objects in the image
if (analysis.Objects.Count > 0)
{
    Console.WriteLine("Objects in image:");

    // Prepare image for drawing
    Image image = Image.FromFile(imageFile);
    Graphics graphics = Graphics.FromImage(image);
    Pen pen = new Pen(Color.Cyan, 3);
    Font font = new Font("Arial", 16);
    SolidBrush brush = new SolidBrush(Color.Black);

    foreach (var detectedObject in analysis.Objects)
    {
        // Print object name
        Console.WriteLine($" -{detectedObject.ObjectProperty} (confidence: {detectedObject.Confidence.T

        // Draw object bounding box
        var r = detectedObject.Rectangle;
        Rectangle rect = new Rectangle(r.X, r.Y, r.W, r.H);
        graphics.DrawRectangle(pen, rect);
        graphics.DrawString(detectedObject.ObjectProperty,font,brush,r.X, r.Y);

    }
    // Save annotated image
    String output_file = "objects.jpg";
    image.Save(output_file);
    Console.WriteLine("  Results saved in " + output_file);
}
```

Python

```
# Get objects in the image
if len(analysis.objects) > 0:
    print("Objects in image:")

    # Prepare image for drawing
    fig = plt.figure(figsize=(8, 8))
    plt.axis('off')
    image = Image.open(image_file)
    draw = ImageDraw.Draw(image)
    color = 'cyan'
    for detected_object in analysis.objects:
        # Print object name
        print(" -{} (confidence: {:.2f}%)".format(detected_object.object_property, detected_object.conf

        # Draw object bounding box
        r = detected_object.rectangle
        bounding_box = ((r.x, r.y), (r.x + r.w, r.y + r.h))
        draw.rectangle(bounding_box, outline=color, width=3)
        plt.annotate(detected_object.object_property,(r.x, r.y), backgroundcolor=color)
    # Save annotated image
    plt.imshow(image)
    outputfile = 'objects.jpg'
    fig.savefig(outputfile)
    print('  Results saved in', outputfile)
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function and observing any objects that are detected. After each run, view the **objects.jpg** file that is generated in the same folder as your code file to see the annotated objects.

20.10 Get moderation ratings for an image

Some images may not be suitable for all audiences, and you may need to apply some moderation to identify images that are adult or violent in nature.

1. In the **AnalyzeImage** function, under the comment **Get moderation ratings**, add the following code:

C#

```
// Get moderation ratings
string ratings = $"Ratings:\n -Adult: {analysis.Adult.IsAdultContent}\n -Racy: {analysis.Adult.IsRacyCo
Console.WriteLine(ratings);
```

Python

```
# Get moderation ratings
ratings = 'Ratings:\n -Adult: {}\n -Racy: {}\n -Gore: {}'.format(analysis.adult.is_adult_content,
                                                                  analysis.adult.is_racy_content,
                                                                  analysis.adult.is_gory_content)
print(ratings)
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function and observing the ratings for each image.

   Note: In the preceding tasks, you used a single method to analyze the image, and then incrementally added code to parse and display the results. The SDK also provides individual methods for suggesting captions, identifying tags, detecting objects, and so on - meaning that you can use the most appropriate method to return only the information you need, reducing the size of the data payload that needs to be returned. See the .NET SDK documentation or Python SDK documentation for more details.

20.11 Generate a thumbnail image

In some cases, you may need to create a smaller version of an image named a *thumbnail*, cropping it to include the main visual subject within new image dimensions.

1. In your code file, find the **GetThumbnail** function; and under the comment **Generate a thumbnail**, add the following code:

C#

```
// Generate a thumbnail
using (var imageData = File.OpenRead(imageFile))
{
    // Get thumbnail data
    var thumbnailStream = await cvClient.GenerateThumbnailInStreamAsync(100, 100,imageData, true);

    // Save thumbnail image
    string thumbnailFileName = "thumbnail.png";
    using (Stream thumbnailFile = File.Create(thumbnailFileName))
    {
        thumbnailStream.CopyTo(thumbnailFile);
    }

    Console.WriteLine($"Thumbnail saved in {thumbnailFileName}");
}
```

Python

```
# Generate a thumbnail
with open(image_file, mode="rb") as image_data:
    # Get thumbnail data
    thumbnail_stream = cv_client.generate_thumbnail_in_stream(100, 100, image_data, True)

# Save thumbnail image
thumbnail_file_name = 'thumbnail.png'
with open(thumbnail_file_name, "wb") as thumbnail_file:
    for chunk in thumbnail_stream:
        thumbnail_file.write(chunk)

print('Thumbnail saved in.', thumbnail_file_name)
```

2. Save your changes and run the program once for each of the image files in the **images** folder, changing the file name in the **Main** function for each run and opening the **thumbnail.jpg** file that is generated in the same folder as your code file for each image.

20.12 More information

In this exercise, you explored some of the image analysis and manipulation capabilities of the Computer Vision service. The service also includes capabilities for reading text, detecting faces, and other computer vision tasks.

20.13 For more information about using the Computer Vision service, see the Computer Vision documentation.

20.14 lab: title: 'Analyze Video with Video Indexer' module: 'Module 8 - Getting Started with Computer Vision'

21 Analyze Video with Video Indexer

A large proportion of the data created and consumed today is in the format of video. **Video Indexer** is an AI-powered service that you can use to index videos and extract insights from them.

21.1 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/MicrosoftLe` 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**.

21.2 Upload a video to Video Indexer

First, you'll need to sign into the Video Indexer portal and upload a video.

> **Tip**: If the Video Indexer page is slow to load in the hosted lab environment, use your locally installed browser. You can switch back to the hosted VM for the later tasks.

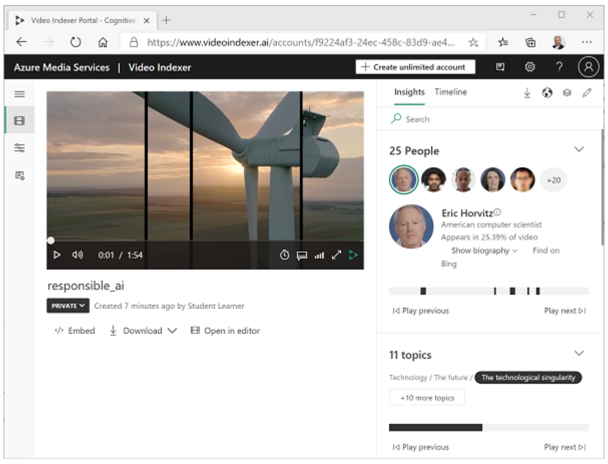
1. In your browser, open the Video Indexer portal at `https://www.videoindexer.ai`.
2. If you have an existing Video Indexer account, sign in. Otherwise, sign up for a free account and sign in using your Microsoft account (or any other valid account type). If you have difficulty signing in, try opening a private browser session.
3. In Video Indexer, select the **Upload** option. Then select the option to **enter a file URL** and enter `https://aka.ms/responsible-ai-video`. Change the default name to **Responsible AI**, review the default settings, select the checkbox to verify compliance with Microsoft's policies for facial recognition, and upload the file.
4. After the file has uploaded, wait a few minutes while Video Indexer automatically indexes it.

> **Note**: In this exercise, we're using this video to explore Video Indexer functionality; but you should take the time to watch it in full when you've finished the exercise as it contains useful information and guidance for developing AI-enabled applications responsibly!

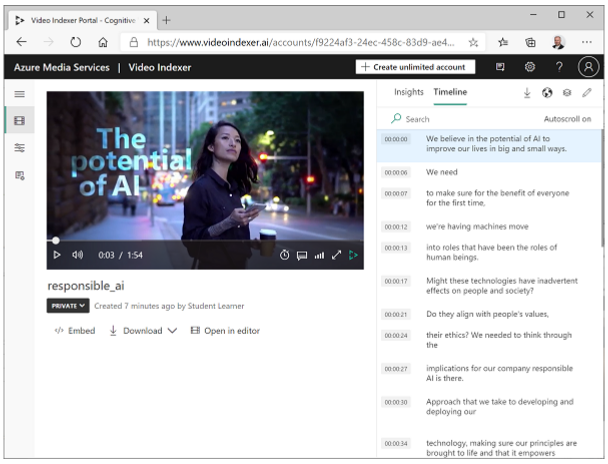
21.3 Review video insights

The indexing process extracts insights from the video, which you can view in the portal.

1. In the Video Indexer portal, when the video is indexed, select it to view it. You'll see the video player alongside a pane that shows insights extracted from the video.

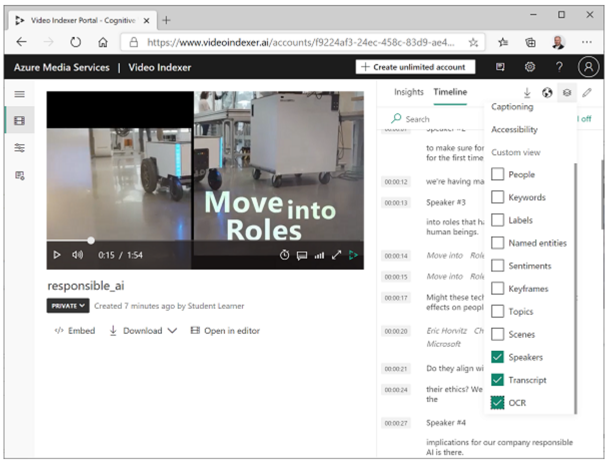

Video Indexer Portal - Cognitive
https://www.videoindexer.ai/accounts/f9224af3-24ec-458c-83d9-ae4...
Azure Media Services | Video Indexer
+ Create unlimited account
Insights
Timeline
Search
25 People
+20
Eric Horvitz
American computer scientist
Appears in 25.39% of video
Show biography
Find on Bing
Play previous
Play next
0:01 / 1:54
responsible_ai
PRIVATE
Created 7 minutes ago by Student Learner
Embed
Download
Open in editor
11 topics
Technology / The future / The technological singularity
+10 more topics
Play previous
Play next

2. As the video plays, select the **Timeline** tab to view a transcript of the video audio.


Video Indexer Portal - Cognitive
https://www.videoindexer.ai/accounts/f9224af3-24ec-458c-83d9-ae4...
Azure Media Services | Video Indexer
Create unlimited account
The potential of AI
0:03 / 1:54
responsible_ai
PRIVATE
Created 7 minutes ago by Student Learner
Embed
Download
Open in editor
Insights
Timeline
Search
Autoscroll on

00:00:00	We believe in the potential of AI to improve our lives in big and small ways.
00:00:06	We need
00:00:07	to make sure for the benefit of everyone for the first time.
00:00:12	we're having machines move
00:00:13	into roles that have been the roles of human beings.
00:00:17	Might these technologies have inadvertent effects on people and society?
00:00:21	Do they align with people's values,
00:00:24	their ethics? We needed to think through the
00:00:27	implications for our company responsible AI is there.
00:00:30	Approach that we take to developing and deploying our
00:00:34	technology, making sure our principles are brought to life and that it empowers

3. At the top right of the portal, select the **View** symbol (which looks similar to), and in the list of insights, in addition to **Transcript**, select **OCR** and **Speakers**.


Video Indexer Portal - Cognitive
https://www.videoindexer.ai/accounts/f9224af3-24ec-458c-83d9-ae4...
Azure Media Services | Video Indexer
Create unlimited account
Move into Roles
0:15 / 1:54
responsible_ai
PRIVATE
Created 7 minutes ago by Student Learner
Embed
Download
Open in editor
Insights
Timeline
Search
Captioning
Accessibility
Custom view

- ☐ People
- ☐ Keywords
- ☐ Labels
- ☐ Named entities
- ☐ Sentiments
- ☐ Keyframes
- ☐ Topics
- ☐ Scenes
- ✓ Speakers
- ✓ Transcript
- ✓ OCR

4. Observe that the **Timeline** pane now includes:

   - Transcript of audio narration.
   - Text visible in the video.
   - Indications of speakers who appear in the video. Some well-known people are automatically recognized by name, others are indicated by number (for example *Speaker #1*).

5. Switch back to the **Insights** pane and view the insights show there. They include:

   - Individual people who appear in the video.
   - Topics discussed in the video.
   - Labels for objects that appear in the video.
   - Named entities, such as people and brands that appear in the video.
   - Key scenes.

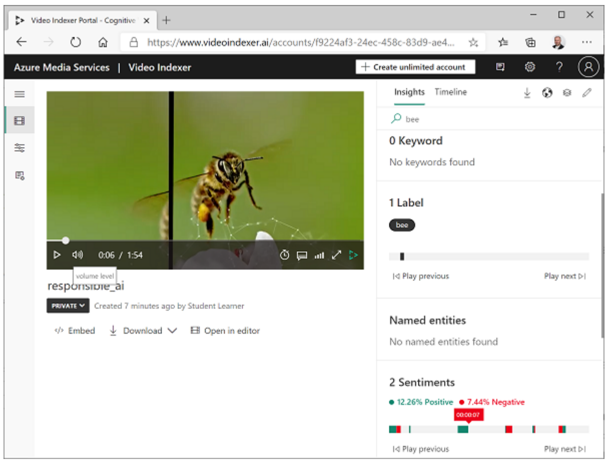
6. With the **Insights** pane visible, select the **View** symbol again, and in the list of insights, add **Keywords** and **Sentiments** to the pane.

   The insights found can help you determine the main themes in the video. For example, the **topics** for this video show that it is clearly about technology, social responsibility, and ethics.

21.4 Search for insights

You can use Video indexer to search the video for insights.

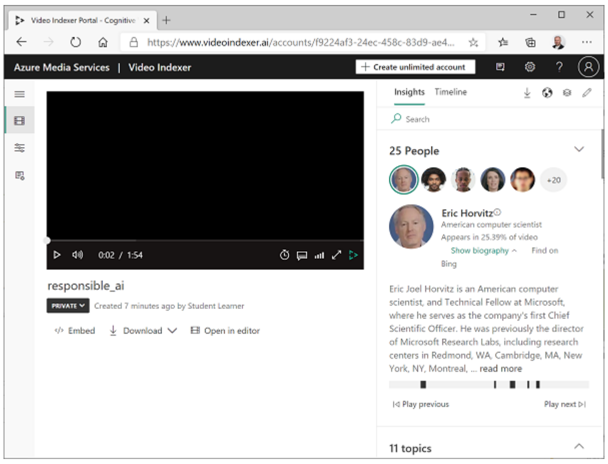
1. In the **Insights** pane, in the **Search** box, enter *Bee*. You may need to scroll down in the Insights pane to see results for all types of insight.
2. Observe that one matching *label* is found, with its location in the video indicated beneath.
3. Select the beginning of the section where the presence of a bee is indicated, and view the video at that point (you may need to pause the video and select carefully - the bee only appears briefly!)
4. Clear the **Search** box to show all insights for the video.



21.5 Edit insights

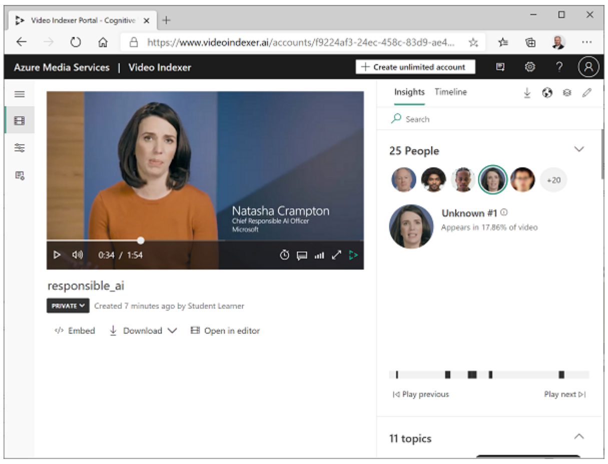
You can use Video Indexer to edit the insights that have been found, adding custom information to make even more sense of the video.

1. Rewind the video to the start and view the **people** listed at the top of the **Insights** pane. Observe that some people have been recognized, including **Eric Horwitz**, a computer scientist and Technical Fellow at Microsoft.


Video Indexer Portal - Cognitive
https://www.videoindexer.ai/accounts/f9224af3-24ec-458c-83d9-ae4...
Azure Media Services | Video Indexer
Create unlimited account
Insights
Timeline
Search
25 People
+20
Eric Horvitz
American computer scientist
Appears in 25.39% of video
Show biography
Find on Bing
Play previous
Play next
11 topics
0:02 / 1:54
responsible_ai
PRIVATE
Created 7 minutes ago by Student Learner
Embed
Download
Open in editor

Eric Joel Horvitz is an American computer scientist, and Technical Fellow at Microsoft, where he serves as the company's first Chief Scientific Officer. He was previously the director of Microsoft Research Labs, including research centers in Redmond, WA, Cambridge, MA, New York, NY, Montreal, ... read more

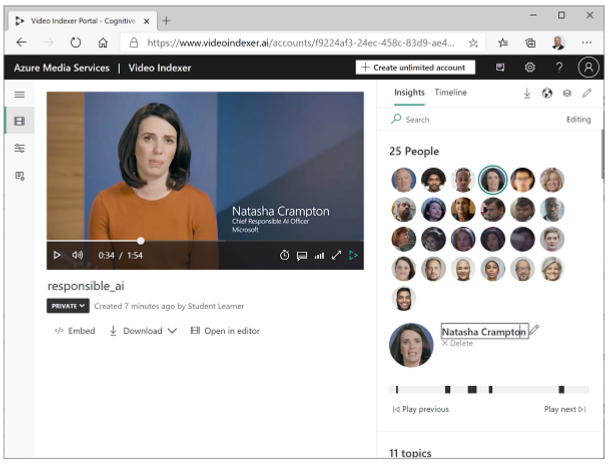
2. Select the photo of Eric Horwitz, and view the information underneath - expanding the **Show biography** section to see information about this person.
3. Observe that the locations in the video where this person appears are indicated. You can use these to view those sections of the video.
4. In the video player, find the person speaking at approximately 0:34:


Video Indexer Portal - Cognitive
https://www.videoindexer.ai/accounts/f9224af3-24ec-458c-83d9-ae4...
Azure Media Services | Video Indexer
Create unlimited account
Insights
Timeline
Search
25 People
+20
Unknown #1
Appears in 17.86% of video
Play previous
Play next
11 topics
Natasha Crampton
Chief Responsible AI Officer
Microsoft
0:34 / 1:54
responsible_ai
PRIVATE
Created 7 minutes ago by Student Learner
Embed
Download
Open in editor

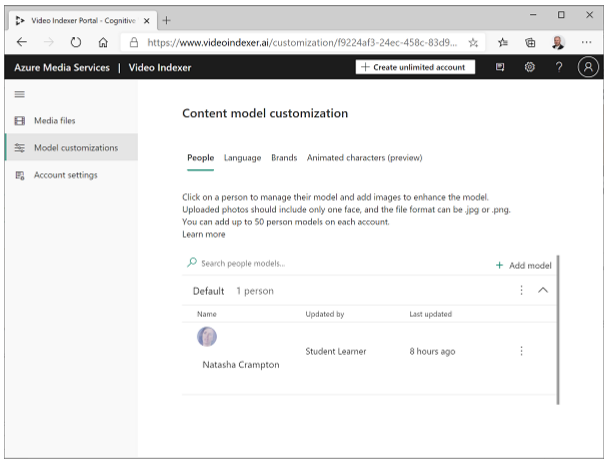
5. Observe that this person is not recognized, and has been assigned a generic name such as **Unknown #1**. However, the video does include a caption with this person's name, so we can enrich the insights by

editing the details for this person.

6. At the top right of the portal, select the **Edit** icon ( ). Then change the name of the unknown person to **Natasha Crampton**.


Azure Media Services | Video Indexer
Insights Timeline
25 People
Natasha Crampton
responsible_ai
Embed Download Open in editor
11 topics

7. After you have made the name change, search the **Insights** pane for *Natasha*. The results should include one person, and indicate the sections of the video in which they appear.
8. At the top left of the portal, expand the menu ( ) and select the **Model customizations** page. Then on the **People** tab, observe that the **Default** people model has one person in it. Video Indexer has added the person you named to a people model, so that they will be recognized in any future videos you index in your account.


Media files
Model customizations
Account settings
Content model customization
People Language Brands Animated characters (preview)

Click on a person to manage their model and add images to enhance the model.
Uploaded photos should include only one face, and the file format can be .jpg or .png.
You can add up to 50 person models on each account.
Learn more

Name	Updated by	Last updated
Natasha Crampton	Student Learner	8 hours ago

You can add images of people to the default people model, or add new models of your own. This enables you to define collections of people with images of their face so that Video Indexer can recognize them in your videos.

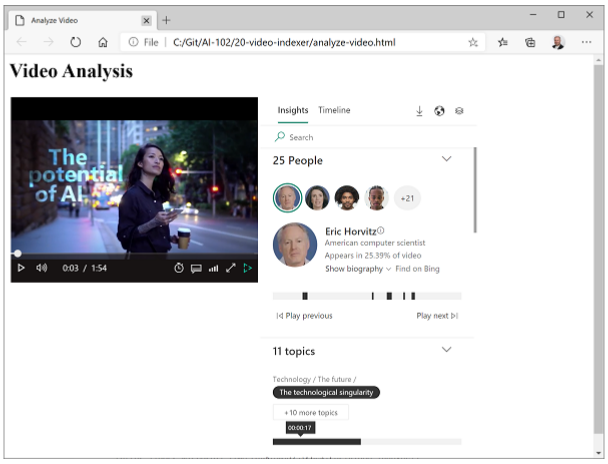
Observe also that you can also create custom models for language (for example to specify industry-specific terminology you want Video Indexer to recognize) and brands (for example, company or product names).

21.6 Use Video Indexer widgets

The Video Indexer portal is a useful interface to manage video indexing projects. However, there may be occasions when you want to make the video and its insights available to people who don't have access to your Video Indexer account. Video Indexer provides widgets that you can embed in a web page for this purpose.

1. In Visual Studio Code, in the **16-video-indexer** folder, open **analyze-video.html**. This is a basic HTML page to which you will add the Video Indexer **Player** and **Insights** widgets. Note the reference to the **vb.widgets.mediator.js** script in the header - this script enables multiple Video Indexer widgets on the page to interact with one another.
2. In the Video Indexer portal, return to the **Media files** page and open your **Responsible AI** video.
3. Under the video player, select **</> Embed** to view the HTML iframe code to embed the widgets.
4. In the **Share and Embed** dialog box, select the **Player** widget, set the video size to 560 x 315, and then copy the embed code to the clipboard.
5. In Visual Studio Code, in the **analyze-video.html** file, paste the copied code under the comment **<-- Player widget goes here -- >**.

6. Back in the **Share and Embed** dialog box, select the **Insights** widget and then copy the embed code to the clipboard. Then close the **Share and Embed** dialog box, switch back to Visual Studio Code, and paste the copied code under the comment **<-- Insights widget goes here -- >**.
7. Save the file. Then in the **Explorer** pane, right-click **analyze-video.html** and select **Reveal in File Explorer**.
8. In File Explorer, open **analyze-video.html** in your browser to see the web page.
9. Experiment with the widgets, using the **Insights** widget to search for insights and jump to them in the video.



21.7 Use the Video Indexer REST API

Video Indexer provides a REST API that you can use to upload and manage videos in your account.

21.7.1 Get your API details

To use the Video Indexer API, you need some information to authenticate requests:

1. In the Video Indexer portal, expand the menu ( ) and select the **Account settings** page.
2. Note the **Account ID** on this page - you will need it later.
3. Open a new browser tab and go to the Video Indexer developer portal at `https://api-portal.videoindexer.ai`, signing in using the credentials for your Video Indexer account.
4. On the **Profile** page, view the **Subscriptions** associated with your profile.
5. On the page with your subscription(s), observe that you have been assigned two keys (primary and secondary) for each subscription. Then select **Show** for any of the keys to see it. You will need this key shortly.

21.7.2 Use the REST API

Now that you have the account ID and an API key, you can use the REST API to work with videos in your account. In this procedure, you'll use a PowerShell script to make REST calls; but the same principles apply with HTTP utilities such as cURL or Postman, or any programming language capable of sending and receiving JSON over HTTP.

All interactions with the Video Indexer REST API follow the same pattern:

- An initial request to the **AccessToken** method with the API key in the header is used to obtain an access token.
- Subsequent requests use the access token to authenticate when calling REST methods to work with videos.

1. In Visual Studio Code, in the **16-video-indexer** folder, open **get-videos.ps1**.
2. In the PowerShell script, replace the **YOUR_ACCOUNT_ID** and **YOUR_API_KEY** placeholders with the account ID and API key values you identified previously.
3. Observe that the *location* for a free account is "trial". If you have created an unrestricted Video Indexer account (with an associated Azure resource), you can change this to the location where your Azure resource is provisioned (for example "eastus").
4. Review the code in the script, noting that invokes two REST methods: one to get an access token, and another to list the videos in your account.
5. Save your changes, and then at the top-right of the script pane, use the   button to run the script.

6. View the JSON response from the REST service, which should contain details of the **Responsible AI** video you indexed previously.

21.8 More information

21.9 For more information about Video Indexer, see the Video Indexer documentation.

21.10 lab: title: 'Classify Images with Custom Vision' module: 'Module 9 - Developing Custom Vision Solutions'

22 Classify Images with Custom Vision

The **Custom Vision** service enables you to create computer vision models that are trained on your own images. You can use it to train *image classification* and *object detection* models; which you can then publish and consume from applications.

In this exercise, you will use the Custom Vision service to train an image classification model that can identify three classes of fruit (apple, banana, and orange).

22.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

22.2 Create Custom Vision resources

Before you can train a model, you will need Azure resources for *training* and *prediction*. You can create **Custom Vision** resources for each of these tasks, or you can create a single **Cognitive Services** resource and use it for either (or both).

In this exercise, you'll create **Custom Vision** resources for training and prediction so that you can manage access and costs for these workloads separately.

1. In a new browser tab, 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 *custom vision*, and create a **Custom Vision** resource with the following settings:
   - **Create options**: 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*
   - **Training location**: *Choose any available region*
   - **Training pricing tier**: F0
   - **Prediction location**: *The same region as the training resource*
   - **Prediction pricing tier**: F0

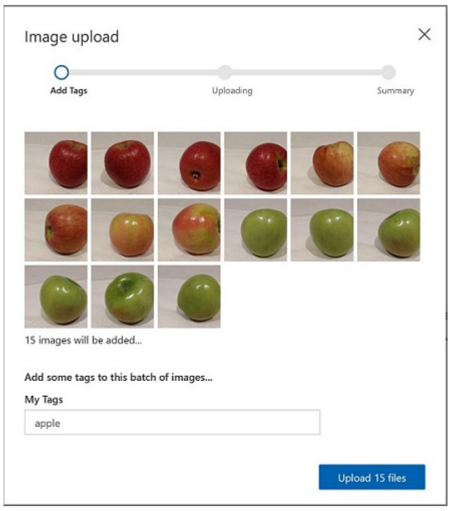
   Note: If you already have an F0 custom vision service in your subscription, select **S0** for this one.
3. Wait for the resources to be created, and then view the deployment details and note that two Custom Vision resources are provisioned; one for training, and another for prediction. You can view these by navigating to the resource group where you created them.

Important: Each resource has its own *endpoint* and *keys*, which are used to manage access from your code. To train an image classification model, your code must use the *training* resource (with its endpoint and key); and to use the trained model to predict image classes, your code must use the *prediction* resource (with its endpoint and key).

22.3 Create a Custom Vision project

To train an image classification model, you need to create a Custom Vision project based on your training resource. To do this, you'll use the Custom Vision portal.

1. In Visual Studio Code, view the training images in the **17-image-classification/training-images** folder where you cloned the repository. This folder contains subfolders of apple, banana, and orange images.
2. In a new browser tab, open the Custom Vision portal at `https://customvision.ai`. If prompted, sign in using the Microsoft account associated with your Azure subscription and agree to the terms of service.
3. In the Custom Vision portal, create a new project with the following settings:
   - **Name**: Classify Fruit
   - **Description**: Image classification for fruit
   - **Resource**: *The Custom Vision resource you created previously*
   - **Project Types**: Classification
   - **Classification Types**: Multiclass (single tag per image)
   - **Domains**: Food
4. In the new project, click [+] **Add images**, and select all of the files in the **training-images/apple** folder you viewed previously. Then upload the image files, specifying the tag *apple*, like this:

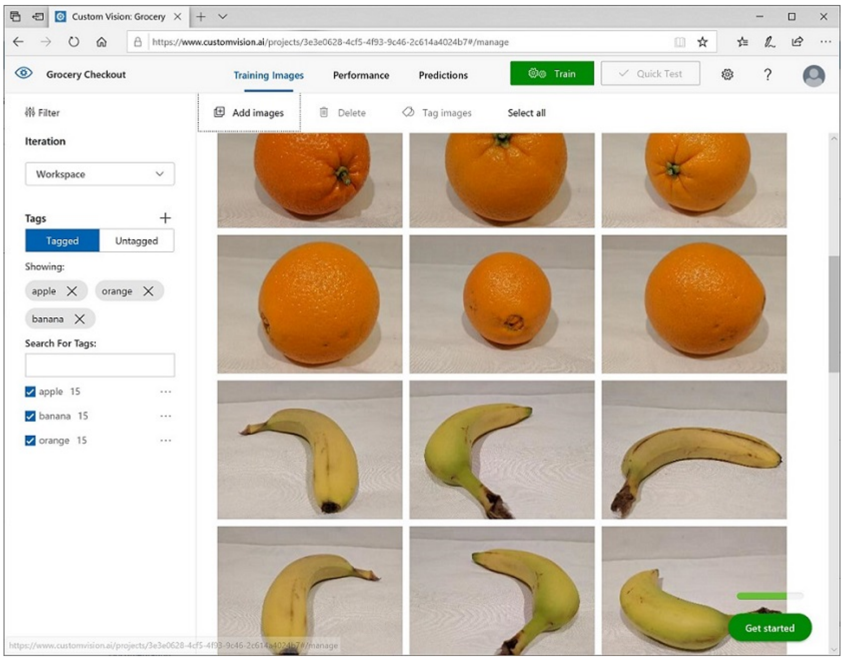


15 images will be added...

Add some tags to this batch of images...

My Tags: apple

5. Repeat the previous step to upload the images in the **banana** folder with the tag *banana*, and the images in the **orange** folder with the tag *orange*.
6. Explore the images you have uploaded in the Custom Vision project - there should be 15 images of each class, like this:



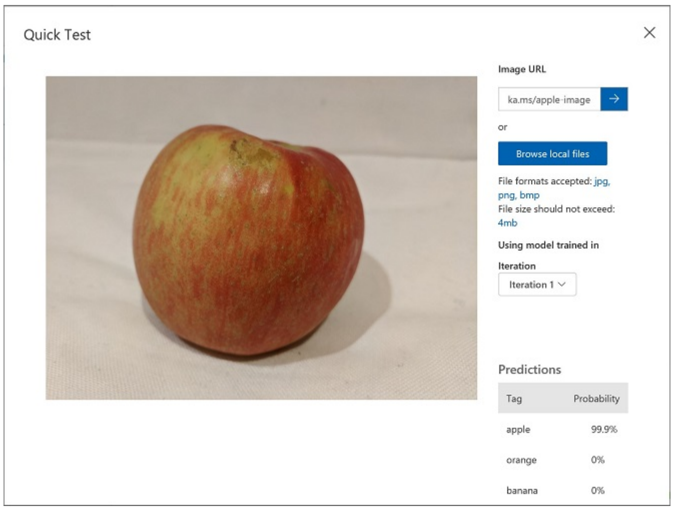
7. In the Custom Vision project, above the images, click **Train** to train a classification model using the tagged images. Select the **Quick Training** option, and then wait for the training iteration to complete (this may take a minute or so).
8. When the model iteration has been trained, review the *Precision*, *Recall*, and *AP* performance metrics - these measure the prediction accuracy of the classification model, and should all be high.

   Note: The performance metrics are based on a probability threshold of 50% for each prediction (in other words, if the model calculates a 50% or higher probability that an image is of a particular class, then that class is predicted). You can adjust this at the top-left of the page.

22.4 Test the model

Now that you've trained the model, you can test it.

1. Above the performance metrics, click **Quick Test**.
2. In the **Image URL** box, type `https://aka.ms/apple-image` and click
3. View the predictions returned by your model - the probability score for *apple* should be the highest, like this:



Predictions

Tag	Probability
apple	99.9%
orange	0%
banana	0%

4. Close the **Quick Test** window.

22.5 View the project settings

The project you have created has been assigned a unique identifier, which you will need to specify in any code that interacts with it.

1. Click the *settings* ( ) icon at the top right of the **Performance** page to view the project settings.
2. Under **General** (on the left), note the **Project Id** that uniquely identifies this project.
3. On the right, under **Resources** note that the details for the *training* resource, including its key and endpoint are shown (you can also obtain this information by viewing the resource in the Azure portal).

22.6 Use the *training* API

The Custom Vision portal provides a convenient user interface that you can use to upload and tag images, and train models. However, in some scenarios you may want to automate model training by using the Custom Vision training API.

> **Note**: In this exercise, you can choose to use the API from either the **C#** or **Python** SDK. In the steps below, perform the actions appropriate for your preferred language.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **17-image_classification** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **train-classifier** folder and open an integrated terminal. Then install the Custom Vision Training package by running the appropriate command for your language preference:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.CustomVision.Training --version 2.0.0
```

Python

```
pip install azure-cognitiveservices-vision-customvision==3.1.0
```

3. View the contents of the **train-classifier** folder, and note that it contains a file for configuration settings:

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

   Open the configuration file and update the configuration values it contains to reflect the endpoint and key for your Custom Vision *training* resource, and the project ID for the classification project you created previously. Save your changes.

4. Note that the **train-classifier** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: train-classifier.py

   Open the code file and review the code it contains, noting the following details:

   - Namespaces from the package you installed are imported
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionTrainingClient**, which is then used with the project ID to create a **Project** reference to your project.
   - The **Upload_Images** function retrieves the tags that are defined in the Custom Vision project and then uploads image files from correspondingly named folders to the project, assigning the appropriate tag ID.
   - The **Train_Model** function creates a new training iteration for the project and waits for training to complete.

5. Return the integrated terminal for the **train-classifier** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python train-classifier.py
```

6. Wait for the program to end. Then return to your browser and view the **Training Images** page for your project in the Custom Vision portal (refreshing the browser if necessary).
7. Verify that some new tagged images have been added to the project. Then view the **Performance** page and verify that a new iteration has been created.

22.7 Publish the image classification model

Now you're ready to publish your trained model so that it can be used from a client application.

1. In the Custom Vision portal, on the **Performance** page, click **Publish** to publish the trained model with the following settings:
   - **Model name**: fruit-classifier
   - **Prediction Resource**: *The **prediction** resource you created previously (not the training resource).*
2. At the top left of the **Project Settings** page, click the *Projects Gallery* ( ) icon to return to the Custom Vision portal home page, where your project is now listed.
3. On the Custom Vision portal home page, at the top right, click the *settings* ( ) icon to view the settings for your Custom Vision service. Then, under **Resources**, find your *prediction* resource (not the training resource) to determine its **Key** and **Endpoint** values (you can also obtain this information by viewing the resource in the Azure portal).

22.8 Use the image classifier from a client application

Now that you've published the image classification model, you can use it from a client application. Once again, you can choose to use **C#** or **Python**.

1. In Visual Studio Code, in the **17-image-classification** folder, in the subfolder for your preferred language (**C-Sharp** or **Python**), right- the **test-classifier** folder and open an integrated terminal. Then enter the following SDK-specific command to install the Custom Vision Prediction package:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.CustomVision.Prediction --version 2.0.0
```

Python

```
pip install azure-cognitiveservices-vision-customvision==3.1.0
```

> **Note**: The Python SDK package includes both training and prediction packages, and may already be installed.

2. Expand the **test-classifier** folder to view the files it contains, which are used to implement a test client application for your image classification model.
3. Open the configuration file for your client application (*appsettings.json* for C# or *.env* for Python) and update the configuration values it contains to reflect the endpoint and key for your Custom Vision *prediction* resource, the project ID for the classification project, and the name of your published model (which should be *fruit-classifier*). Save your changes.
4. Open the code file for your client application (*Program.cs* for C#, *test-classification.py* for Python) and review the code it contains, noting the following details:
   - Namespaces from the package you installed are imported
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionPredictionClient**.
   - The prediction client object is used to predict a class for each image in the **test-images** folder, specifying the project ID and model name for each request. Each prediction includes a probability for each possible class, and only predicted tags with a probability greater than 50% are displayed.
5. Return the integrated terminal for the **test-classifier** folder, and enter the following SDK-specific command to run the program:

C#

```
dotnet run
```

Python

```
python test-classifier.py
```

6. View the label (tag) and probability scores for each prediction. You can view the images in the **test-images** folder to verify that the model has classified them correctly.

22.9 More information

22.10 For more information about image classification with the Custom Vision service, see the Custom Vision documentation.

22.11 lab: title: 'Detect Objects in Images with Custom Vision' module: 'Module 9 - Developing Custom Vision Solutions'

23 Detect Objects in Images with Custom Vision

In this exercise, you will use the Custom Vision service to train an *object detection* model that can detect and locate three classes of fruit (apple, banana, and orange) in an image.

23.1 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/MicrosoftLe` 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**.

23.2 Create Custom Vision resources

If you already have **Custom Vision** resources for training and prediction in your Azure subscription, you can use them in this exercise. If not, use the following instructions to create them.

1. In a new browser tab, 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 *custom vision*, and create a **Custom Vision** resource with the following settings:
   - **Create options**: 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*
   - **Training location**: *Choose any available region*
   - **Training pricing tier**: F0
   - **Prediction location**: *The same region as the training resource*
   - **Prediction pricing tier**: F0

   Note: If you already have an F0 custom vision service in your subscription, select **S0** for this one.
3. Wait for the resources to be created, and then view the deployment details and note that two Custom Vision resources are provisioned; one for training, and another for prediction. You can view these by navigating to the resource group where you created them.

   Important: Each resource has its own *endpoint* and *keys*, which are used to manage access from your code. To train an image classification model, your code must use the *training* resource (with its endpoint and key); and to use the trained model to predict image classes, your code must use the *prediction* resource (with its endpoint and key).

23.3 Create a Custom Vision project

To train an object detection model, you need to create a Custom Vision project based on your training resource. To do this, you'll use the Custom Vision portal.

1. In a new browser tab, open the Custom Vision portal at `https://customvision.ai`, and sign in using the Microsoft account associated with your Azure subscription.
2. Create a new project with the following settings:
   - **Name**: Detect Fruit
   - **Description**: Object detection for fruit.
   - **Resource**: *The Custom Vision resource you created previously*
   - **Project Types**: Object Detection
   - **Domains**: General
3. Wait for the project to be created and opened in the browser.

23.4 Add and tag images

To train an object detection model, you need to upload images that contain the classes you want the model to identify, and tag them to indicate bounding boxes for each object instance.

1. In Visual Studio Code, view the training images in the **18-object-detection/training-images** folder where you cloned the repository. This folder contains images of fruit.
2. In the Custom Vision portal, in your object detection project, select **Add images** and upload all of the images in the extracted folder.
3. After the images have been uploaded, select the first one to open it.
4. Hold the mouse over any object in the image until an automatically detected region is displayed like the image below. Then select the object, and if necessary resize the region to surround it.

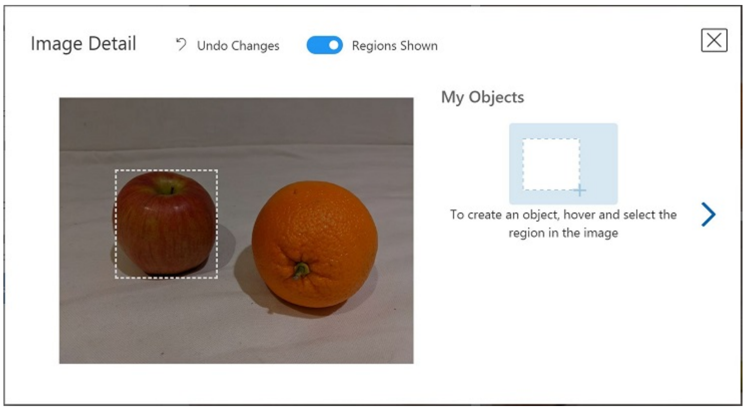


Image Detail
Undo Changes
Regions Shown
My Objects
To create an object, hover and select the region in the image

Alternatively, you can simply drag around the object to create a region.

5. When the region surrounds the object, add a new tag with the appropriate object type (*apple*, *banana*, or *orange*) as shown here:

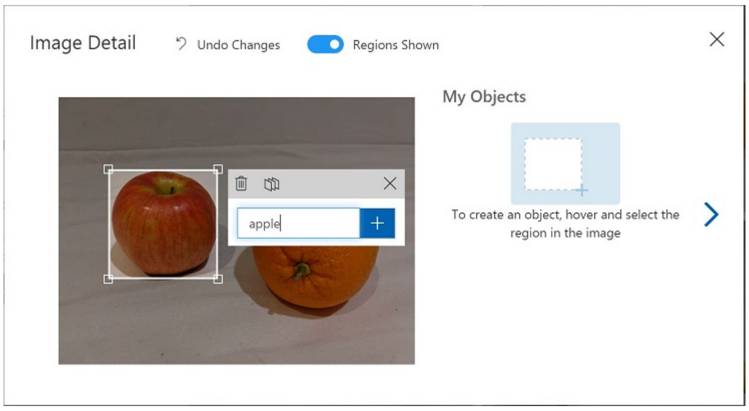
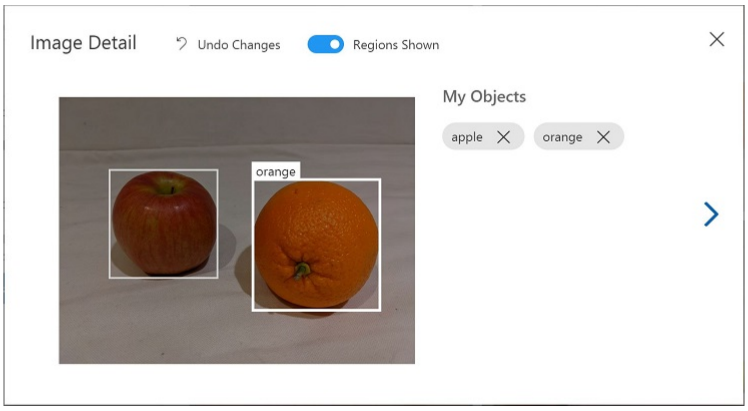


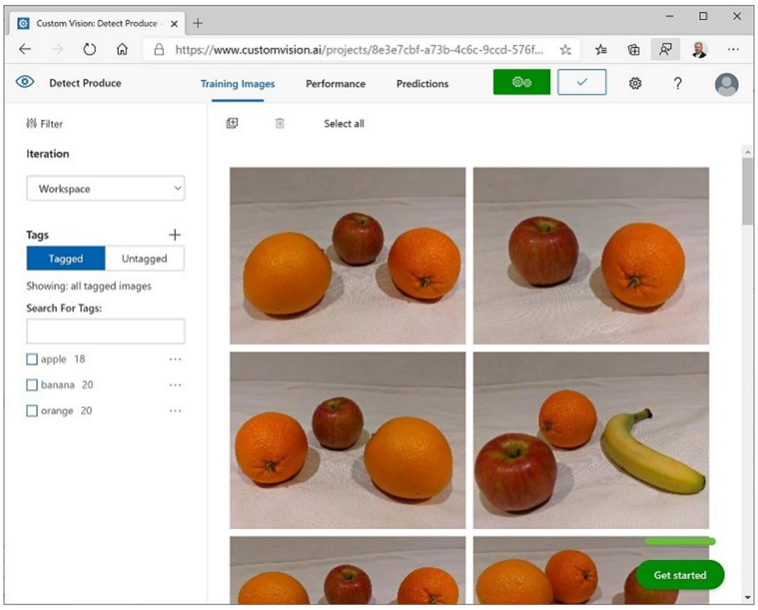
Image Detail
Undo Changes
Regions Shown
apple
My Objects
To create an object, hover and select the region in the image

6. Select and tag each other object in the image, resizing the regions and adding new tags as required.



7. Use the > link on the right to go to the next image, and tag its objects. Then just keep working through the entire image collection, tagging each apple, banana, and orange.

8. When you have finished tagging the last image, close the **Image Detail** editor and on the **Training Images** page, under **Tags**, select **Tagged** to see all of your tagged images:



23.5 Use the Training API to upload images

You can use the graphical tool in the Custom Vision portal to tag your images, but many AI development teams use other tools that generate files containing information about tags and object regions in images. In scenarios like this, you can use the Custom Vision training API to upload tagged images to the project.

> **Note**: In this exercise, you can choose to use the API from either the **C#** or **Python** SDK. In the steps below, perform the actions appropriate for your preferred language.

1. Click the *settings* ( ) icon at the top right of the **Training Images** page in the Custom Vision portal to view the project settings.
2. Under **General** (on the left), note the **Project Id** that uniquely identifies this project.
3. On the right, under **Resources** note that the details for the *training* resource, including its key and endpoint are shown (you can also obtain this information by viewing the resource in the Azure portal).
4. In Visual Studio Code, under the **18-object-detection** folder, expand the **C-Sharp** or **Python** folder depending on your language preference.
5. Right-click the **train-detector** folder and open an integrated terminal. Then install the Custom Vision Training package by running the appropriate command for your language preference:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.CustomVision.Training --version 2.0.0
```

Python

```
pip install azure-cognitiveservices-vision-customvision==3.1.0
```

6. View the contents of the **train-detector** folder, and note that it contains a file for configuration settings:

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

   Open the configuration file and update the configuration values it contains to reflect the endpoint and key for your Custom Vision *training* resource, and the project ID for the classification project you created previously. Save your changes.

7. In the **train-detector** folder, open **tagged-images.json** and examine the JSON it contains. The JSON defines a list of images, each containing one or more tagged regions. Each tagged region includes a tag name, and the top and left coordinates and width and height dimensions of the bounding box containing the tagged object.

   > **Note**: The coordinates and dimensions in this file indicate relative points on the image. For example, a *height* value of 0.7 indicates a box that is 70% of the height of the image. Some tagging tools generate other formats of file in which the coordinate and dimension values represent pixels, inches, or other units of measurements.

8. Note that the **train-detector** folder contains a subfolder in which the image files referenced in the JSON file are stored.

9. Note that the **train-detector** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: train-detector.py

   Open the code file and review the code it contains, noting the following details:

   - Namespaces from the package you installed are imported
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionTrainingClient**, which is then used with the project ID to create a **Project** reference to your project.
   - The **Upload_Images** function extracts the tagged region information from the JSON file and uses it to create a batch of images with regions, which it then uploads to the project.

10. Return the integrated terminal for the **train-detector** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python train-detector.py
```

11. Wait for the program to end. Then return to your browser and view the **Training Images** page for your project in the Custom Vision portal (refreshing the browser if necessary).
12. Verify that some new tagged images have been added to the project.

23.6 Train and test a model

Now that you've tagged the images in your project, you're ready to train a model.

1. In the Custom Vision project, click **Train** to train an object detection model using the tagged images. Select the **Quick Training** option.
2. Wait for training to complete (it might take ten minutes or so), and then review the *Precision*, *Recall*, and *mAP* performance metrics - these measure the prediction accuracy of the classification model, and should all be high.
3. At the top right of the page, click **Quick Test**, and then in the **Image URL** box, enter `https://aka.ms/apple-orange` and view the prediction that is generated. Then close the **Quick Test** window.

23.7 Publish the object detection model

Now you're ready to publish your trained model so that it can be used from a client application.

1. In the Custom Vision portal, on the **Performance** page, click **Publish** to publish the trained model with the following settings:
    - **Model name**: fruit-detector
    - **Prediction Resource**: *The **prediction** resource you created previously (not the training resource).*
2. At the top left of the **Project Settings** page, click the *Projects Gallery* ( ) icon to return to the Custom Vision portal home page, where your project is now listed.
3. On the Custom Vision portal home page, at the top right, click the *settings* ( ) icon to view the settings for your Custom Vision service. Then, under **Resources**, find your *prediction* resource (not the training resource) to determine its **Key** and **Endpoint** values (you can also obtain this information by viewing the resource in the Azure portal).

23.8 Use the image classifier from a client application

Now that you've published the image classification model, you can use it from a client application. Once again, you can choose to use **C#** or **Python**.

1. In Visual Studio Code, browse to the **18-object-detection** folder and in the folder for your preferred language (**C-Sharp** or **Python**), expand the **test-detector** folder.
2. Right-click the **test-detector** folder and open an integrated terminal. Then enter the following SDK-specific command to install the Custom Vision Prediction package:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.CustomVision.Prediction --version 2.0.0
```

Python

```
pip install azure-cognitiveservices-vision-customvision==3.1.0
```

> **Note**: The Python SDK package includes both training and prediction packages, and may already be installed.

3. Open the configuration file for your client application (*appsettings.json* for C# or *.env* for Python) and update the configuration values it contains to reflect the endpoint and key for your Custom Vision *prediction* resource, the project ID for the object detection project, and the name of your published model (which should be *fruit-detector*). Save your changes.
4. Open the code file for your client application (*Program.cs* for C#, *test-detector.py* for Python) and review the code it contains, noting the following details:
    - Namespaces from the package you installed are imported
    - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **CustomVisionPredictionClient**.
    - The prediction client object is used to get object detection predictions for the **produce.jpg** image, specifying the project ID and model name in the request. The predicted tagged regions are then drawn on the image, and the result is saved as **output.jpg**.
5. Return to the integrated terminal for the **test-detector** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python test-detector.py
```

6. After the program has completed, view the resulting **output.jpg** file to see the detected objects in the image.

23.9 More information

23.10 For more information about object detection with the Custom Vision service, see the Custom Vision documentation.

23.11 lab: title: 'Detect, Analyze, and Recognize Faces' module: 'Module 10 - Detecting, Analyzing, and Recognizing Faces'

24 Detect, Analyze, and Recognize Faces

The ability to detect, analyze, and recognize human faces is a core AI capability. In this exercise, you'll explore two Azure Cognitive Services that you can use to work with faces in images: the **Computer Vision** service, and the **Face** service.

24.1 Clone the repository for this course

If you have not already done so, you must clone the code repository for this course:

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

24.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

24.3 Prepare to use the Computer Vision SDK

In this exercise, you'll complete a partially implemented client application that uses the Computer Vision SDK to analyze faces in an image.

> **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 **19-face** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **computer-vision** folder and open an integrated terminal. Then install the Computer Vision SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Microsoft.Azure.CognitiveServices.Vision.ComputerVision --version 6.0.0
   ```

   Python

```
pip install azure-cognitiveservices-vision-computervision==0.7.0
```

3. View the contents of the **computer-vision** folder, and note that it contains a file for configuration settings:
   - **C#**: appsettings.json
   - **Python**: .env
4. Open the configuration file and update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.
5. Note that the **computer-vision** folder contains a code file for the client application:
   - **C#**: Program.cs
   - **Python**: detect-faces.py
6. 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 Computer Vision SDK:

   C#

   ```
   // import namespaces
   using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
   using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
   ```

   Python

   ```
   # import namespaces
   from azure.cognitiveservices.vision.computervision import ComputerVisionClient
   from azure.cognitiveservices.vision.computervision.models import VisualFeatureTypes
   from msrest.authentication import CognitiveServicesCredentials
   ```

24.4 View the image you will analyze

In this exercise, you will use the Computer Vision service to analyze an image of people.

1. In Visual Studio Code, expand the **computer-vision** folder and the **images** folder it contains.
2. Select the **people.jpg** image to view it.

24.5 Detect faces in an image

Now you're ready to use the SDK to call the Computer Vision service and detect faces in an image.

1. In the code file for your client application (**Program.cs** or **detect-faces.py**), in the **Main** function, note that the code to load the configuration settings has been provided. Then find the comment **Authenticate Computer Vision client**. Then, under this comment, add the following language-specific code to create and authenticate a Computer Vision client object:

   C#

   ```
   // Authenticate Computer Vision client
   ApiKeyServiceClientCredentials credentials = new ApiKeyServiceClientCredentials(cogSvcKey);
   cvClient = new ComputerVisionClient(credentials)
   {
       Endpoint = cogSvcEndpoint
   };
   ```

   Python

   ```
   # Authenticate Computer Vision client
   credential = CognitiveServicesCredentials(cog_key)
   cv_client = ComputerVisionClient(cog_endpoint, credential)
   ```

2. In the **Main** function, under the code you just added, note that the code specifies the path to an image file and then passes the image path to a function named **AnalyzeFaces**. This function is not yet fully implemented.
3. In the **AnalyzeFaces** function, under the comment **Specify features to be retrieved (faces)**, add the following code:

C#

```
// Specify features to be retrieved (faces)
List<VisualFeatureTypes?> features = new List<VisualFeatureTypes?>()
{
    VisualFeatureTypes.Faces
};
```

Python

```
# Specify features to be retrieved (faces)
features = [VisualFeatureTypes.faces]
```

4. In the **AnalyzeFaces** function, under the comment **Get image analysis**, add the following code:

C#

```
// Get image analysis
using (var imageData = File.OpenRead(imageFile))
{
    var analysis = await cvClient.AnalyzeImageInStreamAsync(imageData, features);

    // Get faces
    if (analysis.Faces.Count > 0)
    {
        Console.WriteLine($"{analysis.Faces.Count} faces detected.");

        // Prepare image for drawing
        Image image = Image.FromFile(imageFile);
        Graphics graphics = Graphics.FromImage(image);
        Pen pen = new Pen(Color.LightGreen, 3);
        Font font = new Font("Arial", 3);
        SolidBrush brush = new SolidBrush(Color.LightGreen);

        // Draw and annotate each face
        foreach (var face in analysis.Faces)
        {
            var r = face.FaceRectangle;
            Rectangle rect = new Rectangle(r.Left, r.Top, r.Width, r.Height);
            graphics.DrawRectangle(pen, rect);
            string annotation = $"Person aged approximately {face.Age}";
            graphics.DrawString(annotation,font,brush,r.Left, r.Top);
        }

        // Save annotated image
        String output_file = "detected_faces.jpg";
        image.Save(output_file);
        Console.WriteLine(" Results saved in " + output_file);
    }
}
```

Python

```
# Get image analysis
with open(image_file, mode="rb") as image_data:
    analysis = cv_client.analyze_image_in_stream(image_data , features)

    # Get faces
    if analysis.faces:
        print(len(analysis.faces), 'faces detected.')

        # Prepare image for drawing
        fig = plt.figure(figsize=(8, 6))
        plt.axis('off')
        image = Image.open(image_file)
```

```
draw = ImageDraw.Draw(image)
color = 'lightgreen'

# Draw and annotate each face
for face in analysis.faces:
    r = face.face_rectangle
    bounding_box = ((r.left, r.top), (r.left + r.width, r.top + r.height))
    draw = ImageDraw.Draw(image)
    draw.rectangle(bounding_box, outline=color, width=5)
    annotation = 'Person aged approximately {}'.format(face.age)
    plt.annotate(annotation,(r.left, r.top), backgroundcolor=color)

# Save annotated image
plt.imshow(image)
outputfile = 'detected_faces.jpg'
fig.savefig(outputfile)

print('Results saved in', outputfile)
```

5. Save your changes and return to the integrated terminal for the **computer-vision** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python detect-faces.py
   ```

6. Observe the output, which should indicate the number of faces detected.
7. View the **detected_faces.jpg** file that is generated in the same folder as your code file to see the annotated faces. In this case, your code has used the attributes of the face to estimate the age of each person in the image, and the bounding box coordinates to draw a rectangle around each face.

24.6 Prepare to use the Face SDK

While the **Computer Vision** service offers basic face detection (along with many other image analysis capabilities), the **Face** service provides more comprehensive functionality for facial analysis and recognition.

1. In Visual Studio Code, in the **Explorer** pane, browse to the **19-face** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **face-api** folder and open an integrated terminal. Then install the Face SDK package by running the appropriate command for your language preference:

   C#

   ```
   dotnet add package Microsoft.Azure.CognitiveServices.Vision.Face --version 2.6.0-preview.1
   ```

   Python

   ```
   pip install azure-cognitiveservices-vision-face==0.4.1
   ```

3. View the contents of the **face-api** folder, and note that it contains a file for configuration settings:
   - **C#**: appsettings.json
   - **Python**: .env
4. Open the configuration file and update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.
5. Note that the **face-api** folder contains a code file for the client application:
   - **C#**: Program.cs
   - **Python**: analyze-faces.py

6. 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 Computer Vision SDK:

   C#

   ```
   // Import namespaces
   using Microsoft.Azure.CognitiveServices.Vision.Face;
   using Microsoft.Azure.CognitiveServices.Vision.Face.Models;
   ```

   Python

   ```
   # Import namespaces
   from azure.cognitiveservices.vision.face import FaceClient
   from azure.cognitiveservices.vision.face.models import FaceAttributeType
   from msrest.authentication import CognitiveServicesCredentials
   ```

7. In the **Main** function, note that the code to load the configuration settings has been provided. Then find the comment **Authenticate Face client**. Then, under this comment, add the following language-specific code to create and authenticate a **FaceClient** object:

   C#

   ```
   // Authenticate Face client
   ApiKeyServiceClientCredentials credentials = new ApiKeyServiceClientCredentials(cogSvcKey);
   faceClient = new FaceClient(credentials)
   {
       Endpoint = cogSvcEndpoint
   };
   ```

   Python

   ```
   # Authenticate Face client
   credentials = CognitiveServicesCredentials(cog_key)
   face_client = FaceClient(cog_endpoint, credentials)
   ```

8. In the **Main** function, under the code you just added, note that the code displays a menu that enables you to call functions in your code to explore the capabilities of the Face service. You will implement these functions in the remainder of this exercise.

24.7 Detect and analyze faces

One of the most fundamental capabilities of the Face service is to detect faces in an image, and determine their attributes, such as age, emotional expressions, hair color, the presence of spectacles, and so on.

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **1**. This code calls the **DetectFaces** function, passing the path to an image file.

2. Find the **DetectFaces** function in the code file, and under the comment **Specify facial features to be retrieved**, add the following code:

   C#

   ```
   // Specify facial features to be retrieved
   List<FaceAttributeType?> features = new List<FaceAttributeType?>
   {
       FaceAttributeType.Age,
       FaceAttributeType.Emotion,
       FaceAttributeType.Glasses
   };
   ```

   Python

   ```
   # Specify facial features to be retrieved
   features = [FaceAttributeType.age,
               FaceAttributeType.emotion,
               FaceAttributeType.glasses]
   ```

3. In the **DetectFaces** function, under the code you just added, find the comment **Get faces** and add the following code:

C#

```
// Get faces
using (var imageData = File.OpenRead(imageFile))
{
    var detected_faces = await faceClient.Face.DetectWithStreamAsync(imageData, returnFaceAttributes: f

    if (detected_faces.Count > 0)
    {
        Console.WriteLine($"{detected_faces.Count} faces detected.");

        // Prepare image for drawing
        Image image = Image.FromFile(imageFile);
        Graphics graphics = Graphics.FromImage(image);
        Pen pen = new Pen(Color.LightGreen, 3);
        Font font = new Font("Arial", 4);
        SolidBrush brush = new SolidBrush(Color.Black);

        // Draw and annotate each face
        foreach (var face in detected_faces)
        {
            // Get face properties
            Console.WriteLine($"\nFace ID: {face.FaceId}");
            Console.WriteLine($" - Age: {face.FaceAttributes.Age}");
            Console.WriteLine($" - Emotions:");
            foreach (var emotion in face.FaceAttributes.Emotion.ToRankedList())
            {
                Console.WriteLine($"   - {emotion}");
            }

            Console.WriteLine($" - Glasses: {face.FaceAttributes.Glasses}");

            // Draw and annotate face
            var r = face.FaceRectangle;
            Rectangle rect = new Rectangle(r.Left, r.Top, r.Width, r.Height);
            graphics.DrawRectangle(pen, rect);
            string annotation = $"Face ID: {face.FaceId}";
            graphics.DrawString(annotation,font,brush,r.Left, r.Top);
        }

        // Save annotated image
        String output_file = "detected_faces.jpg";
        image.Save(output_file);
        Console.WriteLine(" Results saved in " + output_file);
    }
}
```

Python

```
# Get faces
with open(image_file, mode="rb") as image_data:
    detected_faces = face_client.face.detect_with_stream(image=image_data,
                                                        return_face_attributes=features)

    if len(detected_faces) > 0:
        print(len(detected_faces), 'faces detected.')

        # Prepare image for drawing
        fig = plt.figure(figsize=(8, 6))
        plt.axis('off')
        image = Image.open(image_file)
        draw = ImageDraw.Draw(image)
        color = 'lightgreen'

        # Draw and annotate each face
        for face in detected_faces:

            # Get face properties
            print('\nFace ID: {}'.format(face.face_id))
            detected_attributes = face.face_attributes.as_dict()
            age = 'age unknown' if 'age' not in detected_attributes.keys() else int(detected_attributes
            print(' - Age: {}'.format(age))

            if 'emotion' in detected_attributes:
                print(' - Emotions:')
                for emotion_name in detected_attributes['emotion']:
                    print('   - {}: {}'.format(emotion_name, detected_attributes['emotion'][emotion_nam

            if 'glasses' in detected_attributes:
                print(' - Glasses:{}'.format(detected_attributes['glasses']))

            # Draw and annotate face
            r = face.face_rectangle
            bounding_box = ((r.left, r.top), (r.left + r.width, r.top + r.height))
            draw = ImageDraw.Draw(image)
            draw.rectangle(bounding_box, outline=color, width=5)
            annotation = 'Face ID: {}'.format(face.face_id)
            plt.annotate(annotation,(r.left, r.top), backgroundcolor=color)

        # Save annotated image
        plt.imshow(image)
        outputfile = 'detected_faces.jpg'
        fig.savefig(outputfile)

        print('\nResults saved in', outputfile)
```

4. Examine the code you added to the **DetectFaces** function. It analyzes an image file and detects any faces it contains, including attributes for age, emotions, and the presence of spectacles. The details of each face are displayed, including a unique face identifier that is assigned to each face; and the location of the faces is indicated on the image using a bounding box.

5. Save your changes and return to the integrated terminal for the **face-api** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   *The C# output may display warnings about asynchronous functions now using the **await** operator. You can ignore these.*

   Python

   ```
   python analyze-faces.py
   ```

6. When prompted, enter **1** and observe the output, which should include the ID and attributes of each face detected.

7. View the **detected_faces.jpg** file that is generated in the same folder as your code file to see the annotated faces.

24.8 Compare faces

A common task is to compare faces, and find faces that are similar. In this scenario, you do not need to *identify* the person in the images, just determine whether multiple images show the *same* person (or at least someone who looks similar).

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **2**. This code calls the **CompareFaces** function, passing the path to two image files (**person1.jpg** and **people.jpg**).
2. Find the **CompareFaces** function in the code file, and under the existing code that prints a message to the console, add the following code:

C#

```
// Determine if the face in image 1 is also in image 2
DetectedFace image_i_face;
using (var image1Data = File.OpenRead(image1))
{
    // Get the first face in image 1
    var image1_faces = await faceClient.Face.DetectWithStreamAsync(image1Data);
    if (image1_faces.Count > 0)
    {
        image_i_face = image1_faces[0];
        Image img1 = Image.FromFile(image1);
        Graphics graphics = Graphics.FromImage(img1);
        Pen pen = new Pen(Color.LightGreen, 3);
        var r = image_i_face.FaceRectangle;
        Rectangle rect = new Rectangle(r.Left, r.Top, r.Width, r.Height);
        graphics.DrawRectangle(pen, rect);
        String output_file = "face_to_match.jpg";
        img1.Save(output_file);
        Console.WriteLine(" Results saved in " + output_file);

        //Get all the faces in image 2
        using (var image2Data = File.OpenRead(image2))
        {
            var image2Faces = await faceClient.Face.DetectWithStreamAsync(image2Data);

            // Get faces
            if (image2Faces.Count > 0)
            {

                var image2FaceIds = image2Faces.Select(f => f.FaceId).ToList<Guid?>();
                var similarFaces = await faceClient.Face.FindSimilarAsync((Guid)image_i_face.FaceId,fac
                var similarFaceIds = similarFaces.Select(f => f.FaceId).ToList<Guid?>();

                // Prepare image for drawing
                Image img2 = Image.FromFile(image2);
                Graphics graphics2 = Graphics.FromImage(img2);
                Pen pen2 = new Pen(Color.LightGreen, 3);
                Font font2 = new Font("Arial", 4);
                SolidBrush brush2 = new SolidBrush(Color.Black);

                // Draw and annotate each face
                foreach (var face in image2Faces)
                {
                    if (similarFaceIds.Contains(face.FaceId))
                    {
                        // Draw and annotate face
                        var r2 = face.FaceRectangle;
                        Rectangle rect2 = new Rectangle(r2.Left, r2.Top, r2.Width, r2.Height);
                        graphics2.DrawRectangle(pen2, rect2);
                        string annotation = "Match!";
                        graphics2.DrawString(annotation,font2,brush2,r2.Left, r2.Top);
                    }
                }
                // Save annotated image
                String output_file2 = "matched_faces.jpg";
                img2.Save(output_file2);
                Console.WriteLine(" Results saved in " + output_file2);
            }
        }

    }
}
```

Python

```python
# Determine if the face in image 1 is also in image 2
with open(image_1, mode="rb") as image_data:
    # Get the first face in image 1
    image_1_faces = face_client.face.detect_with_stream(image=image_data)
    image_1_face = image_1_faces[0]

    # Highlight the face in the image
    fig = plt.figure(figsize=(8, 6))
    plt.axis('off')
    image = Image.open(image_1)
    draw = ImageDraw.Draw(image)
    color = 'lightgreen'
    r = image_1_face.face_rectangle
    bounding_box = ((r.left, r.top), (r.left + r.width, r.top + r.height))
    draw = ImageDraw.Draw(image)
    draw.rectangle(bounding_box, outline=color, width=5)
    plt.imshow(image)
    outputfile = 'face_to_match.jpg'
    fig.savefig(outputfile)

# Get all the faces in image 2
with open(image_2, mode="rb") as image_data:
    image_2_faces = face_client.face.detect_with_stream(image=image_data)
    image_2_face_ids = list(map(lambda face: face.face_id, image_2_faces))

    # Find faces in image 2 that are similar to the one in image 1
    similar_faces = face_client.face.find_similar(face_id=image_1_face.face_id, face_ids=image_2_face_i
    similar_face_ids = list(map(lambda face: face.face_id, similar_faces))

    # Prepare image for drawing
    fig = plt.figure(figsize=(8, 6))
    plt.axis('off')
    image = Image.open(image_2)
    draw = ImageDraw.Draw(image)

    # Draw and annotate matching faces
    for face in image_2_faces:
        if face.face_id in similar_face_ids:
            r = face.face_rectangle
            bounding_box = ((r.left, r.top), (r.left + r.width, r.top + r.height))
            draw = ImageDraw.Draw(image)
            draw.rectangle(bounding_box, outline='lightgreen', width=10)
            plt.annotate('Match!',(r.left, r.top + r.height + 15), backgroundcolor='white')

    # Save annotated image
    plt.imshow(image)
    outputfile = 'matched_faces.jpg'
    fig.savefig(outputfile)
```

3. Examine the code you added to the **CompareFaces** function. It finds the first face in image 1 and annotates it in a new image file named **face_to_match.jpg**. Then it finds all of the faces in image 2, and uses their face IDs to find the ones that are similar to image 1. The similar ones are annotated and saved in a new image named **matched_faces.jpg**.
4. Save your changes and return to the integrated terminal for the **face-api** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   *The C# output may display warnings about asynchronous functions now using the **await** operator. You can ignore these.*

   Python

   ```
   python analyze-faces.py
   ```

5. When prompted, enter **2** and observe the output. Then view the **face_to_match.jpg** and **matched_faces.jpg** files that are generated in the same folder as your code file to see the matched faces.
6. Edit the code in the **Main** function for menu option **2** to compare **person2.jpg** to **people.jpg** and then re-run the program and view the results.

24.9 Train a facial recognition model

There may be scenarios where you need to maintain a model of specific people whose faces can be recognized by an AI application. For example, to facilitate a biometric security system that uses facial recognition to verify the identity of employees in a secure building.

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **3**. This code calls the **TrainModel** function, passing the name and ID for a new **PersonGroup** that will be registered in your cognitive services resource, and a list of employee names.
2. In the **face-api/images** folder, observe that there are folders with the same names as the employees. Each folder contains mutliple images of the named employee.
3. Find the **TrainModel** function in the code file, and under the existing code that prints a message to the console, add the following code:

C#

```
// Delete group if it already exists
var groups = await faceClient.PersonGroup.ListAsync();
foreach(var group in groups)
{
    if (group.PersonGroupId == groupId)
    {
        await faceClient.PersonGroup.DeleteAsync(groupId);
    }
}

// Create the group
await faceClient.PersonGroup.CreateAsync(groupId, groupName);
Console.WriteLine("Group created!");

// Add each person to the group
Console.Write("Adding people to the group...");
foreach(var personName in imageFolders)
{
    // Add the person
    var person = await faceClient.PersonGroupPerson.CreateAsync(groupId, personName);

    // Add multiple photo's of the person
    string[] images = Directory.GetFiles("images/" + personName);
    foreach(var image in images)
```

```
    {
        using (var imageData = File.OpenRead(image))
        {
            await faceClient.PersonGroupPerson.AddFaceFromStreamAsync(groupId, person.PersonId, imageDa
        }
    }

}

    // Train the model
Console.WriteLine("Training model...");
await faceClient.PersonGroup.TrainAsync(groupId);

// Get the list of people in the group
Console.WriteLine("Facial recognition model trained with the following people:");
var people = await faceClient.PersonGroupPerson.ListAsync(groupId);
foreach(var person in people)
{
    Console.WriteLine($"-{person.Name}");
}
```

Python

```python
# Delete group if it already exists
groups = face_client.person_group.list()
for group in groups:
    if group.person_group_id == group_id:
        face_client.person_group.delete(group_id)

# Create the group
face_client.person_group.create(group_id, group_name)
print ('Group created!')

# Add each person to the group
print('Adding people to the group...')
for person_name in image_folders:
    # Add the person
    person = face_client.person_group_person.create(group_id, person_name)

    # Add multiple photo's of the person
    folder = os.path.join('images', person_name)
    person_pics = os.listdir(folder)
    for pic in person_pics:
        img_path = os.path.join(folder, pic)
        img_stream = open(img_path, "rb")
        face_client.person_group_person.add_face_from_stream(group_id, person.person_id, img_stream)

# Train the model
print('Training model...')
face_client.person_group.train(group_id)

# Get the list of people in the group
print('Facial recognition model trained with the following people:')
people = face_client.person_group_person.list(group_id)
for person in people:
    print('-', person.name)
```

4. Examine the code you added to the **TrainModel** function. It performs the following tasks:
    - Gets a list of **PersonGroup**s registered in the service, and deletes the specified one if it exists.
    - Creates a group with the specified ID and name.
    - Adds a person to the group for each name specified, and adds the multiple images of each person.
    - Trains a facial recognition model based on the named people in the group and their face images.

- Retrieves a list of the named people in the group and displays them.

5. Save your changes and return to the integrated terminal for the **face-api** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   *The C# output may display warnings about asynchronous functions now using the **await** operator. You can ignore these.*

   Python

   ```
   python analyze-faces.py
   ```

6. When prompted, enter **3** and observe the output, noting that the **PersonGroup** created includes two people.

24.10 Recognize faces

Now that you have defined a **PeopleGroup** and trained a facial recognition model, you can use it to recognize named individuals in an image.

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **4**. This code calls the **RecognizeFaces** function, passing the path to an image file (**people.jpg**) and the ID of the **PeopleGroup** to be used for face identification.
2. Find the **RecognizeFaces** function in the code file, and under the existing code that prints a message to the console, add the following code:

C#

```
// Detect faces in the image
using (var imageData = File.OpenRead(imageFile))
{
    var detectedFaces = await faceClient.Face.DetectWithStreamAsync(imageData);

    // Get faces
    if (detectedFaces.Count > 0)
    {

        // Get a list of face IDs
        var faceIds = detectedFaces.Select(f => f.FaceId).ToList<Guid?>();

        // Identify the faces in the people group
        var recognizedFaces = await faceClient.Face.IdentifyAsync(faceIds, groupId);

        // Get names for recognized faces
        var faceNames = new Dictionary<Guid?, string>();
        if (recognizedFaces.Count> 0)
        {
            foreach(var face in recognizedFaces)
            {
                var person = await faceClient.PersonGroupPerson.GetAsync(groupId, face.Candidates[0].Pe
                Console.WriteLine($"-{person.Name}");
                faceNames.Add(face.FaceId, person.Name);

            }
        }


        // Annotate faces in image
        Image image = Image.FromFile(imageFile);
        Graphics graphics = Graphics.FromImage(image);
        Pen penYes = new Pen(Color.LightGreen, 3);
        Pen penNo = new Pen(Color.Magenta, 3);
```

```
        Font font = new Font("Arial", 4);
        SolidBrush brush = new SolidBrush(Color.Cyan);
        foreach (var face in detectedFaces)
        {
            var r = face.FaceRectangle;
            Rectangle rect = new Rectangle(r.Left, r.Top, r.Width, r.Height);
            if (faceNames.ContainsKey(face.FaceId))
            {
                // If the face is recognized, annotate in green with the name
                graphics.DrawRectangle(penYes, rect);
                string personName = faceNames[face.FaceId];
                graphics.DrawString(personName,font,brush,r.Left, r.Top);
            }
            else
            {
                // Otherwise, just annotate the unrecognized face in magenta
                graphics.DrawRectangle(penNo, rect);
            }
        }

        // Save annotated image
        String output_file = "recognized_faces.jpg";
        image.Save(output_file);
        Console.WriteLine("Results saved in " + output_file);
    }
}
```

Python

```
# Detect faces in the image
with open(image_file, mode="rb") as image_data:

    # Get faces
    detected_faces = face_client.face.detect_with_stream(image=image_data)

    # Get a list of face IDs
    face_ids = list(map(lambda face: face.face_id, detected_faces))

    # Identify the faces in the people group
    recognized_faces = face_client.face.identify(face_ids, group_id)

    # Get names for recognized faces
    face_names = {}
    if len(recognized_faces) > 0:
        print(len(recognized_faces), 'faces recognized.')
        for face in recognized_faces:
            person_name = face_client.person_group_person.get(group_id, face.candidates[0].person_id).n
            print('-', person_name)
            face_names[face.face_id] = person_name

    # Annotate faces in image
    fig = plt.figure(figsize=(8, 6))
    plt.axis('off')
    image = Image.open(image_file)
    draw = ImageDraw.Draw(image)
    for face in detected_faces:
        r = face.face_rectangle
        bounding_box = ((r.left, r.top), (r.left + r.width, r.top + r.height))
        draw = ImageDraw.Draw(image)
        if face.face_id in face_names:
            # If the face is recognized, annotate in green with the name
            draw.rectangle(bounding_box, outline='lightgreen', width=3)
            plt.annotate(face_names[face.face_id],
                         (r.left, r.top + r.height + 15), backgroundcolor='white')
        else:
            # Otherwise, just annotate the unrecognized face in magenta
            draw.rectangle(bounding_box, outline='magenta', width=3)

    # Save annotated image
    plt.imshow(image)
    outputfile = 'recognized_faces.jpg'
    fig.savefig(outputfile)

    print('\nResults saved in', outputfile)
```

3. Examine the code you added to the **RecognizeFaces** function. It finds the faces in the image and creates a list of their IDs. Then it uses the people group to try to identify the faces in the list of face IDs. The recognized faces are annotated wuth the name of the identified person, and the results are saved in **recognized_faces.jpg**.
4. Save your changes and return to the integrated terminal for the **face-api** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   *The C# output may display warnings about asynchronous functions now using the **await** operator. You can ignore these.*

   Python

   ```
   python analyze-faces.py
   ```

5. When prompted, enter **4** and observe the output. Then view the **recognized_faces.jpg** file that is generated in the same folder as your code file to see the identified people.
6. Edit the code in the **Main** function for menu option **4** to to recognize faces in **people2.jpg** and then re-run the program and view the results. The same people should be recognized.

24.11 Verify a face

Facial recognition is often used for identity verification. With the Face service, you can verify a face in an image by comparing it to another face, or from a person registered in a **PersonGroup**.

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **5**. This code calls the **VerifyFace** function, passing the path to an image file (**person1.jpg**), a name, and the ID of the **PeopleGroup** to be used for face identification.
2. Find the **VerifyFace** function in the code file, and under the comment **Get the ID of the person from the people group** (above the code that prints the result) add the following code:

C#

```
// Get the ID of the person from the people group
var people = await faceClient.PersonGroupPerson.ListAsync(groupId);
foreach(var person in people)
{
    if (person.Name == personName)
    {
        Guid personId = person.PersonId;

        // Get the first face in the image
        using (var imageData = File.OpenRead(personImage))
        {
            var faces = await faceClient.Face.DetectWithStreamAsync(imageData);
            if (faces.Count > 0)
            {
                Guid faceId = (Guid)faces[0].FaceId;
```

```
                //We have a face and an ID. Do they match?
                var verification = await faceClient.Face.VerifyFaceToPersonAsync(faceId, personId, grou
                if (verification.IsIdentical)
                {
                    result = "Verified";
                }
            }
        }
    }
}
```

Python

```python
# Get the ID of the person from the people group
people = face_client.person_group_person.list(group_id)
for person in people:
    if person.name == person_name:
        person_id = person.person_id

        # Get the first face in the image
        with open(person_image, mode="rb") as image_data:
            faces = face_client.face.detect_with_stream(image=image_data)
            if len(faces) > 0:
                face_id = faces[0].face_id

                # We have a face and an ID. Do they match?
                verification = face_client.face.verify_face_to_person(face_id, person_id, group_id)
                if verification.is_identical:
                    result = 'Verified'
```

3. Examine the code you added to the **VerifyFace** function. It looks up the ID of the person in the group with the specified name. If the person exists, the code gets the face ID of the first face in the image. Finally, if there is a face in the image, the code verifies it against the ID of the specified person.
4. Save your changes and return to the integrated terminal for the **face-api** folder, and enter the following command to run the program:

   C#

   ```
   dotnet run
   ```

   Python

   ```
   python analyze-faces.py
   ```

5. When prompted, enter **5** and observe the result.
6. Edit the code in the **Main** function for menu option **5** to to experiment with different combinations of names and the images **person1.jpg** and **person2.jpg**.

24.12 More information

For more information about using the **Computer Vision** service for face detection, see the Computer Vision documentation.

24.13 To learn more about the Face service, see the Face documentation.

24.14 lab: title: 'Read Text in Images' module: 'Module 11 - Reading Text in Images and Documents'

25 Read Text in Images

Optical character recognition (OCR) is a subset of computer vision that deals with reading text in images and documents. The **Computer Vision** service provides two APIs for reading text, which you'll explore in this exercise.

25.1 Clone the repository for this course

If you have not already done so, you must clone the code repository for this course:

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

25.2 Provision a Cognitive Services resource

If you don't already have one in your subscription, you'll need to provision a **Cognitive Services** resource.

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 *cognitive services*, and create a **Cognitive Services** resource with the following settings:
   - **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)*
   - **Region**: *Choose any available region*
   - **Name**: *Enter a unique name*
   - **Pricing tier**: Standard S0
3. Select the required checkboxes and create the resource.
4. Wait for deployment to complete, and then view the deployment details.
5. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the endpoint and one of the keys from this page in the next procedure.

25.3 Prepare to use the Computer Vision SDK

In this exercise, you'll complete a partially implemented client application that uses the Computer Vision SDK to read text.

> **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 **20-ocr** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **read-text** folder and open an integrated terminal. Then install the Computer Vision SDK package by running the appropriate command for your language preference:

C#

```
dotnet add package Microsoft.Azure.CognitiveServices.Vision.ComputerVision --version 6.0.0
```

Python

```
pip install azure-cognitiveservices-vision-computervision==0.7.0
```

3. View the contents of the **read-text** folder, and note that it contains a file for configuration settings:

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

   Open the configuration file and update the configuration values it contains to reflect the **endpoint** and an authentication **key** for your cognitive services resource. Save your changes.

4. Note that the **read-text** folder contains a code file for the client application:

   - **C#**: Program.cs
   - **Python**: read-text.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 Computer Vision SDK:

C#

```
// import namespaces
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision;
using Microsoft.Azure.CognitiveServices.Vision.ComputerVision.Models;
```

Python

```
# import namespaces
from azure.cognitiveservices.vision.computervision import ComputerVisionClient
from azure.cognitiveservices.vision.computervision.models import OperationStatusCodes
from msrest.authentication import CognitiveServicesCredentials
```

5. In the code file for your client application, in the **Main** function, note that the code to load the configuration settings has been provided. Then find the comment **Authenticate Computer Vision client**. Then, under this comment, add the following language-specific code to create and authenticate a Computer Vision client object:

C#

```
// Authenticate Computer Vision client
ApiKeyServiceClientCredentials credentials = new ApiKeyServiceClientCredentials(cogSvcKey);
cvClient = new ComputerVisionClient(credentials)
{
    Endpoint = cogSvcEndpoint
};
```

Python

```
# Authenticate Computer Vision client
credential = CognitiveServicesCredentials(cog_key)
cv_client = ComputerVisionClient(cog_endpoint, credential)
```

25.4 Use the OCR API

The **OCR** API is an optical character recognition API that is optimized for reading small to medium amounts of printed text in *.jpg*, *.png*, *.gif*, and *.bmp* format images. It supports a wide range of languages and in addition to reading text in the image it can determine the orientation of each text region and return information about the rotation angle of the text in relation to the image

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **1**. This code calls the **GetTextOcr** function, passing the path to an image file.
2. In the **read-text/images** folder, open **Lincoln.jpg** to view the image that your code will process.
3. Back in the code file, find the **GetTextOcr** function, and under the existing code that prints a message to the console, add the following code:

C#

```
// Use OCR API to read text in image
using (var imageData = File.OpenRead(imageFile))
{
    var ocrResults = await cvClient.RecognizePrintedTextInStreamAsync(detectOrientation:false, image:im

    // Prepare image for drawing
    Image image = Image.FromFile(imageFile);
    Graphics graphics = Graphics.FromImage(image);
    Pen pen = new Pen(Color.Magenta, 3);

    foreach(var region in ocrResults.Regions)
    {
        foreach(var line in region.Lines)
        {
            // Show the position of the line of text
            int[] dims = line.BoundingBox.Split(",").Select(int.Parse).ToArray();
            Rectangle rect = new Rectangle(dims[0], dims[1], dims[2], dims[3]);
            graphics.DrawRectangle(pen, rect);

            // Read the words in the line of text
            string lineText = "";
            foreach(var word in line.Words)
            {
                lineText += word.Text + " ";
            }
            Console.WriteLine(lineText.Trim());
        }
    }

    // Save the image with the text locations highlighted
    String output_file = "ocr_results.jpg";
    image.Save(output_file);
    Console.WriteLine("Results saved in " + output_file);
}
```

Python

```
# Use OCR API to read text in image
with open(image_file, mode="rb") as image_data:
    ocr_results = cv_client.recognize_printed_text_in_stream(image_data)

# Prepare image for drawing
fig = plt.figure(figsize=(7, 7))
img = Image.open(image_file)
draw = ImageDraw.Draw(img)

# Process the text line by line
for region in ocr_results.regions:
    for line in region.lines:

        # Show the position of the line of text
        l,t,w,h = list(map(int, line.bounding_box.split(',')))
        draw.rectangle(((l,t), (l+w, t+h)), outline='magenta', width=5)

        # Read the words in the line of text
        line_text = ''
        for word in line.words:
            line_text += word.text + ' '
        print(line_text.rstrip())

# Save the image with the text locations highlighted
plt.axis('off')
plt.imshow(img)
outputfile = 'ocr_results.jpg'
fig.savefig(outputfile)
print('Results saved in', outputfile)
```

4. Examine the code you added to the **GetTextOcr** function. It detects regions of printed text from an image file, and for each region it extracts the lines of text and highlights there position on the image. It then extracts the words in each line and prints them.
5. Save your changes and return to the integrated terminal for the **read-text** folder, and enter the following command to run the program:

C#

```
dotnet run
```

*The C# output may display warnings about asynchronous functions now using the **await** operator. You can ignore these.*

Python

```
python read-text.py
```

6. When prompted, enter **1** and observe the output, which is the text extracted from the image.
7. View the **ocr_results.jpg** file that is generated in the same folder as your code file to see the annotated lines of text in the image.

25.5 Use the Read API

The **Read** API uses a newer text recognition model than the OCR API, and performs better for larger images that contain a lot of text. It also supports text extraction from *.pdf* files, and can recognize both printed text (in multiple languages) and handwritten text (in English).

The **Read** API uses an asynchronous operation model, in which a request to start text recognition is submitted; and the operation ID returned from the request can subsequently be used to check progress and retrieve results.

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **2**. This code calls the **GetTextRead** function, passing the path to a PDF document file.
2. In the **read-text/images** folder, right-click **Rome.pdf** and select **Reveal in File Explorer**. Then in File Explorer, open the PDF file to view it.
3. Back in the code file in Visual Studio Code, find the **GetTextRead** function, and under the existing code that prints a message to the console, add the following code:

C#

```
// Use Read API to read text in image
using (var imageData = File.OpenRead(imageFile))
{
    var readOp = await cvClient.ReadInStreamAsync(imageData);

    // Get the async operation ID so we can check for the results
    string operationLocation = readOp.OperationLocation;
    string operationId = operationLocation.Substring(operationLocation.Length - 36);

    // Wait for the asynchronous operation to complete
    ReadOperationResult results;
    do
    {
        Thread.Sleep(1000);
        results = await cvClient.GetReadResultAsync(Guid.Parse(operationId));
    }
    while ((results.Status == OperationStatusCodes.Running ||
            results.Status == OperationStatusCodes.NotStarted));

    // If the operation was successfuly, process the text line by line
    if (results.Status == OperationStatusCodes.Succeeded)
    {
        var textUrlFileResults = results.AnalyzeResult.ReadResults;
        foreach (ReadResult page in textUrlFileResults)
        {
            foreach (Line line in page.Lines)
            {
                Console.WriteLine(line.Text);
            }
        }
    }
}
```

Python

```
# Use Read API to read text in image
with open(image_file, mode="rb") as image_data:
    read_op = cv_client.read_in_stream(image_data, raw=True)
    # Get the async operation ID so we can check for the results
    operation_location = read_op.headers["Operation-Location"]
    operation_id = operation_location.split("/")[-1]

    # Wait for the asynchronous operation to complete
    while True:
        read_results = cv_client.get_read_result(operation_id)
        if read_results.status not in [OperationStatusCodes.running, OperationStatusCodes.not_started]:
            break
        time.sleep(1)

    # If the operation was successfuly, process the text line by line
    if read_results.status == OperationStatusCodes.succeeded:
        for page in read_results.analyze_result.read_results:
            for line in page.lines:
                print(line.text)
```

4. Examine the code you added to the **GetTextRead** function. It submits a request for a read operation, and then repeatedly checks status until the operation has completed. If it was successful, the code processes the results by iterating through each page, and then through each line.
5. Save your changes and return to the integrated terminal for the **read-text** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python read-text.py
```

6. When prompted, enter **2** and observe the output, which is the text extracted from the document.

25.6 Read handwritten text

In addition to printed text, the **Read** API can extract handwritten text in English..

1. In the code file for your application, in the **Main** function, examine the code that runs if the user selects menu option **3**. This code calls the **GetTextRead** function, passing the path to an image file.
2. In the **read-text/images** folder, open **Note.jpg** to view the image that your code will process.
3. In the integrated terminal for the **read-text** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python read-text.py
```

4. When prompted, enter **3** and observe the output, which is the text extracted from the document.

25.7 More information

25.8 For more information about using the Computer Vision service to read text, see the Computer Vision documentation.

25.9 lab: title: 'Extract Data from Forms' module: 'Module 11 - Reading Text in Images and Documents'

26 Extract Data from Forms

Suppose a company needs to automate a data entry process. Currently an employee might manually read a purchase order and enter the data into a database. You want to build a model that will use machine learning to read the form and produce structured data that can be used to automatically update a database.

Form Recognizer is a cognitive service that enables users to build automated data processing software. This software can extract text, key/value pairs, and tables from form documents using optical character recognition (OCR). Form Recognizer has pre-built models for recognizing invoices, receipts, and business cards. The service also provides the capability to train custom models. In this exercise, we will focus on building custom models.

26.1 Clone the repository for this course

If you have not already done so, you must clone the code repository for this course:

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

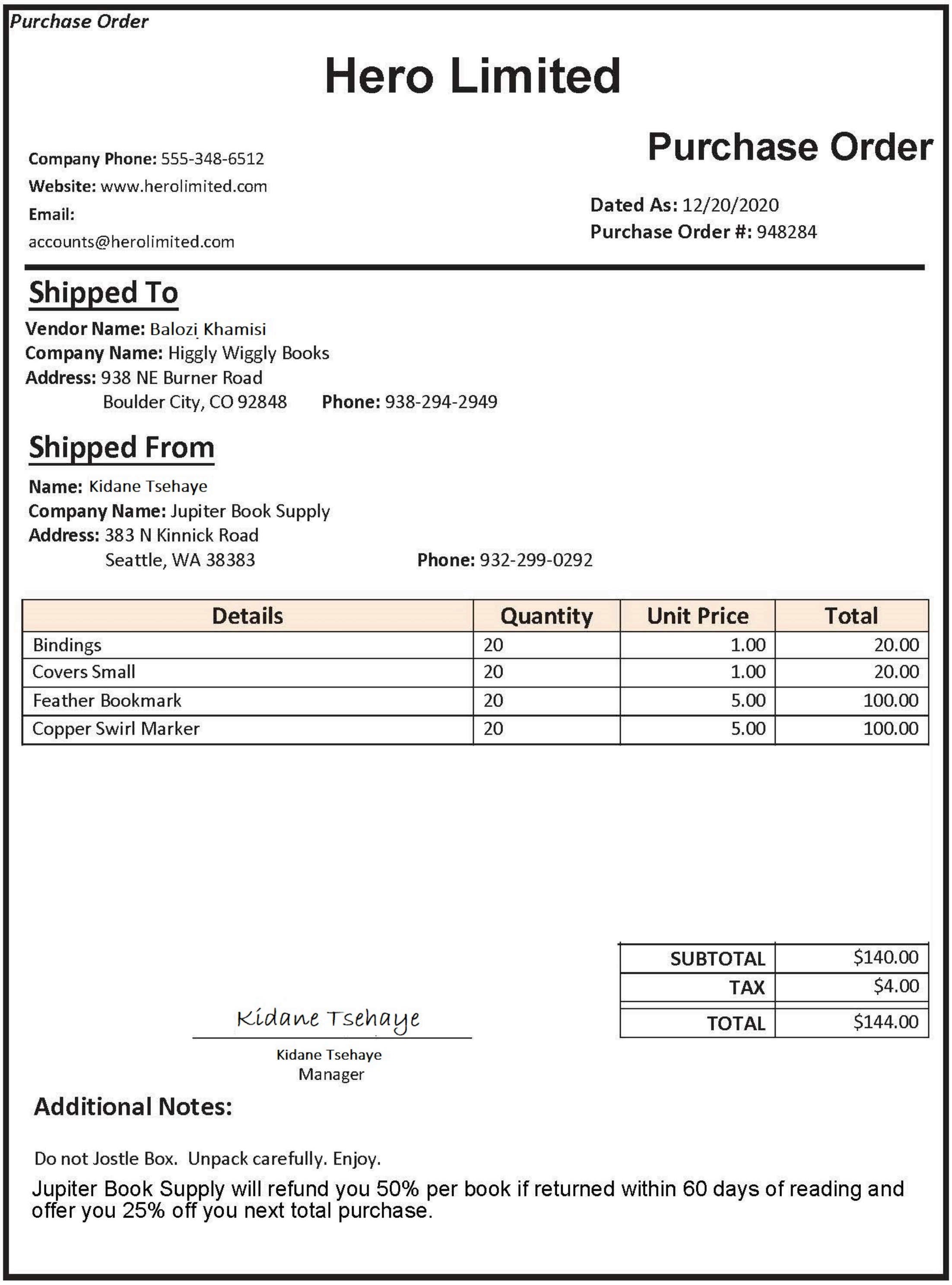
26.2 Create a Form Recognizer resource

To use the Form Recognizer service, you need a Form Recognizer resource in your Azure subscription. You'll use the Azure portal to create a resource.

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 *Form Recognizer*, and create a **Form Recognizer** resource with the following settings:
    - **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)*
    - **Region**: *Choose any available region*
    - **Name**: *Enter a unique name*
    - **Pricing tier**: F0

    Note: If you already have an F0 form recognizer service in your subscription, select **S0** for this one.
3. When the resource has been deployed, go to it and view its **Keys and Endpoint** page. You will need the **endpoint** and one of the **keys** from this page to manage access from your code later on.

26.3 Gather documents for training



Purchase Order

Hero Limited

Purchase Order

Company Phone: 555-348-6512
Website: www.herolimited.com
Email: accounts@herolimited.com

Dated As: 12/20/2020
Purchase Order #: 948284

Shipped To

Vendor Name: Balozi Khamisi
Company Name: Higgly Wiggly Books
Address: 938 NE Burner Road
Boulder City, CO 92848 **Phone:** 938-294-2949

Shipped From

Name: Kidane Tsehaye
Company Name: Jupiter Book Supply
Address: 383 N Kinnick Road
Seattle, WA 38383 **Phone:** 932-299-0292

Details	Quantity	Unit Price	Total
Bindings	20	1.00	20.00
Covers Small	20	1.00	20.00
Feather Bookmark	20	5.00	100.00
Copper Swirl Marker	20	5.00	100.00

SUBTOTAL	$140.00
TAX	$4.00
TOTAL	$144.00

Kidane Tsehaye

Kidane Tsehaye
Manager

Additional Notes:

Do not Jostle Box. Unpack carefully. Enjoy.

Jupiter Book Supply will refund you 50% per book if returned within 60 days of reading and offer you 25% off you next total purchase.

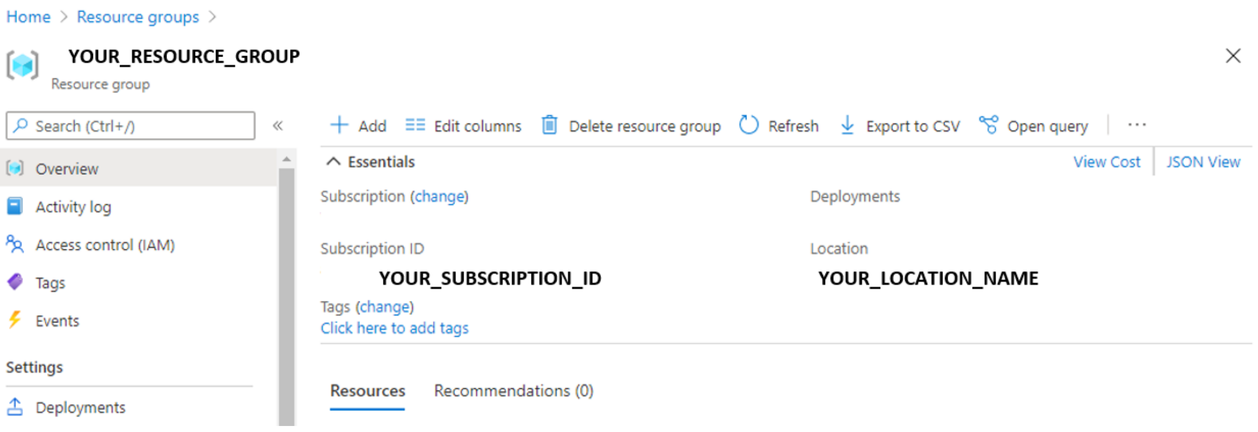
You'll use the sample forms from the **21-custom-form/sample-forms** folder in this repo, which contain all the files you'll need to train a model without labels and another model with labels.

1. In Visual Studio Code, in the **21-custom-form** folder, expand the **sample-forms** folder. Notice there are files ending in **.json** and **.jpg** in the folder.

   You will use the **.jpg** files to train your first model *without* labels.

Later, you will use the files ending in **.json** and **.jpg** to train your second model *with* labels. The **.json** files have been generated for you and contain label information. To train with labels, you need to have the label information files in your blob storage container alongside the forms.

2. Return to the Azure portal at https://portal.azure.com.
3. View the **Resource group** in which you created the Form Recognizer resource previously.
4. On the **Overview** page for your resource group, note the **Subscription ID** and **Location**. You will need these values, along with your **resource group** name in subsequent steps.



5. In Visual Studio Code, in the Explorer pane, right-click the the **21-custom-form** folder and select **Open in Integrated Terminal**.
6. In the terminal pane, enter the following command to establish an authenticated connection to your Azure subscription.

```
az login --output none
```

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

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

9. In the output, find the **Name** value that corresponds with the location of your resource group (for example, for *East US* the corresponding name is *eastus*).

   Important: Record the **Name** value and use it in Step 12.

10. In the Explorer pane, in the **21-custom-form** folder, select **setup.cmd**. You will use this batch script to run the Azure command line interface (CLI) commands required to create the other Azure resources you need.
11. In the **setup.cmd** script, review the **rem** commands. These comments outline the program the script will run. The program will:
    - Create a storage account in your Azure resource group
    - Upload files from your local *sampleforms* folder to a container called *sampleforms* in the storage account
    - Print a Shared Access Signature URI
12. Modify the **subscription_id**, **resource_group**, and **location** variable declarations with the appropriate values for the subscription, resource group, and location name where you deployed the Form Recognizer resource. Then **save** your changes.

    Leave the **expiry_date** variable as it is for the exercise. This variable is used when generating the Shared Access Signature (SAS) URI. In practice, you will want to set an appropriate expiry date for your SAS. You can learn more about SAS here.

13. In the terminal for the **21-custom-form** folder, enter the following command to run the script:

```
setup
```

14. When the script completes, review the displayed output and note your Azure resource's SAS URI.

Important: Before moving on, paste the SAS URI somewhere you will be able to retrieve it again later (for example, in a new text file in Visual Studio Code).

15. In the Azure portal, refresh the resource group and verify that it contains the Azure Storage account just created. Open the storage account and in the pane on the left, select **Storage Explorer**. Then in Storage Explorer, expand **BLOB CONTAINERS** and select the **sampleforms** container to verify that the files have been uploaded from your local **21-custom-form/sample-forms** folder.

26.4 Train a model *without* labels

You will use the Form Recognizer SDK to train and test a custom model.

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

1. In Visual Studio Code, in the **21-custom-form** folder, expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **train-model** folder and open an integrated terminal.
3. Install the Form Recognizer package by running the appropriate command for your language preference:

C#

```
dotnet add package Azure.AI.FormRecognizer --version 3.0.0
```

Python

```
pip install azure-ai-formrecognizer==3.0.0
```

3. View the contents of the **train-model** folder, and note that it contains a file for configuration settings:
   - **C#**: appsettings.json
   - **Python**: .env
4. Edit the configuration file, modifying the settings to reflect:
   - The **endpoint** for your Form Recognizer resource.
   - A **key** for your Form Recognizer resource.
   - The **SAS URI** for your blob container.
5. Note that the **train-model** folder contains a code file for the client application:
   - **C#**: Program.cs
   - **Python**: train-model.py

   Open the code file and review the code it contains, noting the following details:
   - Namespaces from the package you installed are imported
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **Client**.
   - The code uses the the training client to train a model using the images in your blob storage container, which is acessed using the SAS URI you generated.
   - Training is performed with a parameter to indicate that training labels should not be used. Form Recognizer uses an *unsupervised* technique to extract the fields from the form images.
6. Return the integrated terminal for the **train-model** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python train-model.py
```

7. Wait for the program to end. Then review the model output and locate the Model ID in the terminal. You will need this value in the next procedure, so do not close the terminal!

26.5 Test the model created without labels

Now you're ready use your trained model. Notice how you trained your model using files from a storage container URI. You could also have trained the model using local files. Similarly, you can test your model using forms from a URI or from local files. You will test the form model with a local file.

Now that you've got the model ID, you can use it from a client application. Once again, you can choose to use **C#** or **Python**.

1. In the **21-custom-form** folder, in the subfolder for your preferred language (**C-Sharp** or **Python**), expand the **test-model** folder.
2. Right-click the **test-model** folder and open an integrated terminal. You now have (at least) two **cmd** terminals, and you can switch between them using the drop-down list in the Terminal pane.
3. In the terminal for the **test-model** folder, install the Form Recognizer package by running the appropriate command for your language preference:

C#

```
dotnet add package Azure.AI.FormRecognizer --version 3.0.0
```

Python

```
pip install azure-ai-formrecognizer==3.0.0
```

This isn't strictly necessary if you previously used pip to install the package into Python environment; but it does no harm to ensure it's installed!

4. In the **test-model** folder, edit the configuration file (**appsettings.json** or **.env**, depending on your language preference) to add the following values:
   - Your Form Recognizer endpoint.
   - Your Form Recognizer key.
   - The Model ID generated when you trained the model (you can find this by switching the terminal back to the **cmd** console for the **train-model** folder).
5. In the **test-model** folder, open the code file for your client application (*Program.cs* for C#, *test-model.py* for Python) and review the code it contains, noting the following details:
   - Namespaces from the package you installed are imported
   - The **Main** function retrieves the configuration settings, and uses the key and endpoint to create an authenticated **Client**.
   - The client is then used to extract form fields and values from the **test1.jpg** image.
6. Return the integrated terminal for the **test-model** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python test-model.py
```

7. View the output and notice the prediction confidence scores. Notice how the output provides field names field-1, field-2 etc.

26.6 Train a model *with* labels using the client library

Suppose after you trained a model with the invoice forms, you wanted to see how a model trained on labeled data performs. When you trained a model without labels you only used the **.jpg** forms from your Azure blob container. Now you will train a model using the **.jpg** and **.json** files.

1. In Visual Studio Code, in the **21-custom-form/sample-forms** folder, open **fields.json** and review the JSON document it contains. This file defines the fields that you will train a model to extract from the forms.
2. Open **Form_1.jpg.labels.json** and review the JSON it contains. This file identifies the location and values for named fields in the **Form_1.jpg** training document.

3. Open **Form_1.jpg.ocr.json** and review the JSON it contains. This file contains a JSOn representation of the text layout of **Form_1.jpg**, including the location of all text areas found in the form.

   The field information files have been provided for you in this exercise. For your own projects, you can create these files using the sample labeling tool. As you use the tool, your field information files are automatically created and stored in your connected storage account.

4. In the **train-model** folder, open the code file for the training application:
   - **C#**: Program.cs
   - **Python**: train-model.py
5. In the **Main** function, find the comment **Train model**, and modify it as shown to change the training process so that labels are used:

C#

```
// Train model
CustomFormModel model = await trainingClient
.StartTrainingAsync(new Uri(trainingStorageUri), useTrainingLabels: true)
.WaitForCompletionAsync();
```

Python

```
# Train model
poller = form_training_client.begin_training(trainingDataUrl, use_training_labels=True)
model = poller.result()
```

6. Return the integrated terminal for the **train-model** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python train-model.py
```

10. Wait for the program to end, then review the model output.
11. Note the new the Model ID in the terminal output.

26.7 Test the model created with labels

1. In the **test-model** folder, edit the configuration file (**appsettings.json** or **.env**, depending on your language preference) and update it to reflect the new model ID. Save your changes.
2. Return the integrated terminal for the **test-model** folder, and enter the following command to run the program:

C#

```
dotnet run
```

Python

```
python test-model.py
```

3. View the output and observe how the output for the model trained **with** labels provides field names like ”CompanyPhoneNumber” and ”DatedAs” unlike the output from the model trained **without** labels, which produced an output of field-1, field-2 etc.

While the program code for training a model *with* labels may not differ greatly from the code for training *without* labels, choosing one versus the other *does* change project planning needs. To train with labels, you will need to create the labeled files. The choice of training process can also produce different models, which can in turn affect downstream processes based on what fields the model returns and how confident you are with the returned values.

26.8 More information

26.9 For more information about the Form Recognizer service, see the Form Recognizer documentation.

26.10 lab: title: 'Create an Azure Cognitive Search solution' module: 'Module 12 - Creating a Knowledge Mining Solution'

27 Create an Azure Cognitive Search solution

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

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

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

27.1 Clone the repository for this course

If you have not already cloned **AI-102-AIEngineer** code repository to the environment where you're working on this lab, follow these steps to do so. Otherwise, open the cloned folder in Visual Studio Code.

1. Start Visual Studio Code.
2. Open the palette (SHIFT+CTRL+P) and run a **Git: Clone** command to clone the `https://github.com/MicrosoftLe` 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**.

27.2 Create Azure resources

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

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

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

27.2.1 Create an Azure Cognitive Search resource

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

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

27.2.2 Create a Cognitive Services resource

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

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

27.2.3 Create a storage account

1. Return to the home page of the Azure portal, and then select the **Create a resource** button, search for *storage account*, and create a **Storage account** resource with the following settings:

    - **Subscription**: *Your Azure subscription*
    - **Resource group**: **The same resource group as your Azure Cognitive Search and Cognitive Services resources*
    - **Storage account name**: *Enter a unique name*
    - **Location**: *Choose any available location*
    - **Performance**: Standard
    - **Account kind**: Storage V2
    - **Replication**: Locally-redundant storage (LRS)

2. Wait for deployment to complete, and then go to the deployed resource.

3. On the **Overview** page, note the **Subscription ID** -this identifies the subscription in which the storage account is provisioned.

4. On the **Access keys** page, note that two keys have been generated for your storage account. Then select **Show keys** to view the keys.

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

27.3 Upload Documents to Azure Storage

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

1. In Visual Studio Code, in the **Explorer** pane, expand the **22-create-a-search-solution** folder and select **UploadDocs.cmd**.

2. Edit the batch file to replace the **YOUR_SUBSCRIPTION_ID**, **YOUR_AZURE_STORAGE_ACCOUNT** and **YOUR_AZURE_STORAGE_KEY** placeholders with the appropriate subscription ID, Azure storage account name, and Azure storage account key values for the storage account you created previously.

3. Save your changes, and then right-click the **22-create-a-search-solution** folder and open an integrated terminal.

4. Enter the following command to sign into your Azure subscription by using the Azure CLI.

    ```
    az login
    ```

A web browser tab will open and prompt you to sign into Azure. Do so, and then close the browser tab and return to Visual Studio Code.

5. Enter the following command to run the batch file. This will create a blob container in your storage account and upload the documents in the **data** folder to it.

```
UploadDocs
```

27.4 Index the documents

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

1. In the Azure portal, browse to your Azure Cognitive Search resource. Then, on its **Overview** page, select **Import data**.
2. On the **Connect to your data** page, in the **Data Source** list, select **Azure Blob Storage**. Then complete the data store details with the following values:
   - **Data Source**: Azure Blob Storage
   - **Data source name**: margies-data
   - **Data to extract**: Content and metadata
   - **Parsing mode**: Default
   - **Connection string**: *Select **Choose an existing connection**. Then select your storage account, and finally select the **margies** container that was created by the UploadDocs.cmd script.*
   - **Authenticate using managed identity**: Unselected
   - **Container name**: margies
   - **Blob folder**: *Leave this blank*
   - **Description**: Brochures and reviews in Margie's Travel web site.
3. Proceed to the next step (*Add cognitive skills*).
4. in the **Attach Cognitive Services** section, select your cognitive services resource.
5. In the **Add enrichments** section:
   - Change the **Skillset name** to **margies-skillset**.
   - Select the option **Enable OCR and merge all text into merged_content field**.
   - Ensure that the **Source data field** is set to **merged_content**.
   - Leave the **Enrichment granularity level** as **Source field**, which is set the entire contents of the document being indexed; but note that you can change this to extract information at more granular levels, like pages or sentences.
   - Select the following enriched fields:

Cognitive Skill	Parameter	Field name
Extract location names		locations
Extract key phrases		keyphrases
Detect language		language
Generate tags from images		imageTags
Generate captions from images		imageCaption

6. Double-check your selections (it can be difficult to change them later). Then proceed to the next step (*Customize target index*).
7. Change the **Index name** to **margies-index**.
8. Ensure that the **Key** is set to **metadata_storage_path** and leave the **Suggester name** and **Search mode** blank.
9. Make the following changes to the index fields, leaving all other fields with their default settings (**IMPORTANT**: you may need to scroll to the right to see the entire table):

Field name	Retrievable	Filterable	Sortable	Facetable	Searchable
metadata_storage_size					
metadata_storage_last_modified					
metadata_storage_name					
metadata_author					
locations					
keyphrases					

Field name	Retrievable	Filterable	Sortable	Facetable	Searchable
language					

10. Double-check your selections, paying particular attention to ensure that the correct **Retrievable**, **Filterable**, **Sortable**, **Facetable**, and **Searchable** options are selected for each field (it can be difficult to change them later). Then proceed to the next step (*Create an indexer*).
11. Change the **Indexer name** to **margies-indexer**.
12. Leave the **Schedule** set to **Once**.
13. Expand the **Advanced** options, and ensure that the **Base-64 encode keys** option is selected (generally encoding keys make the index more efficient).
14. Select **Submit** to create the data source, skillset, index, and indexer. The indexer is run automatically and runs the indexing pipeline, which:
    1. Extracts the document metadata fields and content from the data source
    2. Runs the skillset of cognitive skills to generate additional enriched fields
    3. Maps the extracted fields to the index.
15. In the bottom half of the **Overview** page for your Azure Cognitive Search resource, view the **Indexers** tab, which should show the newly created **margies-indexer**. Wait a few minutes, and click **Refresh** until the **Status** indicates success.

27.5 Search the index

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

1. At the top of the **Overview** page for your Azure Cognitive Search resource, select **Search explorer**.
2. In Search explorer, in the **Query string** box, enter `*` (a single asterisk), and then select **Search**.

   This query retrieves all documents in the index in JSON format. Examine the results and note the fields for each document, which contain document content, metadata, and enriched data extracted by the cognitive skills you selected.
3. Modify the query string to `search=*&$count=true` and submit the search.

   This time, the results include a **@odata.count** field at the top of the results that indicates the number of documents returned by the search.
4. Try the following query string:

   ```
   search=*&$count=true&$select=metadata_storage_name,metadata_author,locations
   ```

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

   ```
   search="New York"&$count=true&$select=metadata_storage_name,keyphrases
   ```

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

   ```
   search="New York"&$count=true&$select=metadata_storage_name&$filter=metadata_author eq 'Reviewer'
   ```

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

27.6 Explore and modify definitions of search components

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

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

27.6.1 Get the endpoint and key for your Azure Cognitive Search resource

1. In the Azure portal, return to the **Overview** page for your Azure Cognitive Search resource; and in the top section of the page, find the **Url** for your resource (which looks like **https://resource_name.search.windows.net**) and copy it to the clipboard.
2. In Visual Studio Code, in the Explorer pane, expand the **22-create-a-search-solution** folder and its **modify-search** subfolder, and select **modify-search.cmd** to open it. You will use this script file to run *cURL* commands that submit JSON to the Azure Cognitive Service REST interface.
3. In **modify-search.cmd**, replace the **YOUR_SEARCH_URL** placeholder with the URL you copied to the clipboard.
4. In the Azure portal, view the **Keys** page for your Azure Cognitive Search resource, and copy the **Primary admin key** to the clipboard.
5. In Visual Studio Code, replace the **YOUR_ADMIN_KEY** placeholder with the key you copied to the clipboard.
6. Save the changes to **modify-search.cmd** (but don't run it yet!)

27.6.2 Review and modify the skillset

1. In Visual studio Code, in the **modify-search** folder, open **skillset.json**. This shows a JSON definition for **margies-skillset**.

2. At the top of the skillset definition, note the **cognitiveServices** object, which is used to connect your Cognitive Services resource to the skillset.

3. In the Azure portal, open your Cognitive Services resource (not your Azure Cognitive Search resource!) and view its **Keys** page. Then copy **Key 1** to the clipboard.

4. In Visual Studio Code, in **skillset.json**, replace the **YOUR_COGNITIVE_SERVICES_KEY** placeholder with the Cognitive Services key you copied to the clipboard.

5. Scroll through the JSON file, noting that it includes definitions for the skills you created using the Azure Cognitive Search user interface in the Azure portal. At the bottom of the list of skills, an additional skill has been added with the following definition:

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

The new skill is named **get-sentiment**, and for each **document** level in a document, it, will evaluate the text found in the **merged_content** field of the document being indexed (which includes the source content as well as any text extracted from images in the content). It uses the extracted **language** of the document (with a default of English), and evaluates a score for the sentiment of the content. This score is then output as a new field named **sentimentScore**.

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

27.6.3 Review and modify the index

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

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

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

27.6.4 Review and modify the indexer

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

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

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

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

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

27.6.5 Use the REST API to update the search solution

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

   ```
   modify-search
   ```

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

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

27.6.6 Query the modified index

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

   ```
   search=London&$select=url,sentiment,keyphrases&$filter=metadata_author eq 'Reviewer' and sentiment
   ```

   This query retrieves the **url**, **sentiment**, and **keyphrases** for all documents that mention *London* authored by *Reviewer* that have a **sentiment** score greater than *0.5* (in other words, positive reviews that mention London)

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

27.7 Create a search client application

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

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

27.7.1 Get the endpoint and keys for your search resource

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

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

27.7.2 Prepare to use the Azure Cognitive Search SDK

1. In Visual Studio Code, in the **Explorer** pane, browse to the **22-create-a-search-solution** folder and expand the **C-Sharp** or **Python** folder depending on your language preference.
2. Right-click the **margies-travel** folder and open an integrated terminal. Then install the Azure Cognitive Search SDK package by running the appropriate command for your language preference:

   C#

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

   Python

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

3. View the contents of the **margies-travel** folder, and note that it contains a file for configuration settings:

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

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

27.7.3 Explore code to search an index

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

1. Open the following code file in the web application, depending on your choice of programming language:
   - **C#**:Pages/Index.cshtml.cs
   - **Python**: app.py
2. Near the top of the code file, find the comment **Import search namespaces**, and note the namespaces that have been imported to work with the Azure Cognitive Search SDK:
3. In the **search_query** function, find the comment **Create a search client**, and note that the code creates a **SearchClient** object using the endpoint and query key for your Azure Cognitive Search resource:
4. In the **search_query** function, find the comment **Submit search query**, and review the code to submit a search for the specified text with the following options:
   - A *search mode* that requires **all** of the individual words in the search text are found.
   - The total number of documents found by the search is included in the results.
   - The results are filtered to include only documents that match the provided filter expression.
   - The results are sorted into the specified sort order.
   - Each discrete value of the **metadata_author** field is returned as a *facet* that can be used to display pre-defined values for filtering.
   - Up to three extracts of the **merged_content** and **imageCaption** fields with the search terms highlighted are included in the results.
   - The results include only the fields specified.

27.7.4 Explore code to render search results

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

1. Open the following code file in the web application, depending on your choice of programming language:
   - **C#**:Pages/Index.cshtml
   - **Python**: templates/search.html
2. Examine the code, which renders the page on which the search results are displayed. Observe that:
   - The page begins with a search form that the user can use to submit a new search (in the Python version of the application, this form is defined in the **base.html** template), which is referenced at the beginning of the page.
   - A second form is then rendered, enabling the user to refine the search results. The code for this form:
     - Retrieves and displays the count of documents from the search results.
     - Retrieves the facet values for the **metadata_author** field and displays them as an option list for filtering.
     - Creates a drop-down list of sort options for the results.
   - The code then iterates through the search results, rendering each result as follows:
     - Display the **metadata_storage_name** (file name) field as a link to the address in the **url** field.
     - Displaying *highlights* for search terms found in the **merged_content** and **imageCaption** fields to help show the search terms in context.
     - Display the **metadata_author**, **metadata_storage_size**, **metadata_storage_last_modified**, and **language** fields.
     - Indicate the **sentiment** using an emoticon (for scores of 0.5 or higher, and   for scores less than 0.5).
     - Display the first five **keyphrases** (if any).
     - Display the first five **locations** (if any).
     - Display the first five **imageTags** (if any).

27.7.5 Run the web app

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

C#

```
dotnet run
```

Python

```
flask run
```

2. In the message that is displayed when the app starts successfully, follow the link to the running web application (*http://localhost:5000/* or *http://127.0.0.1:5000/*) to open the Margies Travel site in a web browser.
3. In the Margie's Travel website, enter **London hotel** into the search box and click **Search**.
4. Review the search results. They include the file name (with a hyperlink to the file URL), an extract of the file content with the search terms (*London* and *hotel*) emphasized, and other attributes of the file from the index fields.
5. Observe that the results page includes some user interface elements that enable you to refine the results. These include:
   - A *filter* based on a facet value for the **metadata_author** field. This demonstrates how you can use *facetable* fields to return a list of *facets* - fields with a small set of discrete values that can displayed as potential filter values in the user interface.
   - The ability to *order* the results based on a specified field and sort direction (ascending or descending). The default order is based on *relevancy*, which is calculated as a **search.score()** value based on a *scoring profile* that evaluates the frequency and importance of search terms in the index fields.
6. Select the **Reviewer** filter and the **Positive to negative** sort option, and then select **Refine Results**.
7. Observe that the results are filtered to include only reviews, and sorted into descending order of sentiment.
8. In the **Search** box, enter a new search for **quiet hotel in New York** and review the results.
9. Try the following search terms:
   - **Tower of London** (observe that this term is identified as a *key phrase* in some documents).
   - **skyscraper** (observe that this word doesn't appear in the actual content of any documents, but is found in the *image captions* and *image tags* that were generated for images in some documents).
   - **Mojave desert** (observe that this term is identified as a *location* in some documents).
10. Close the browser tab containing the Margie's Travel web site and return to Visual Studio Code. Then in the Python terminal for the **margies-travel** folder (where the dotnet or flask application is running), enter Ctrl+C to stop the app.

27.8 More information

27.9 To learn more about Azure Cognitive Search, see the Azure Cognitive Search documentation.

27.10 lab: title: 'Create a Custom Skill for Azure Cognitive Search' module: 'Module 12 - Creating a Knowledge Mining Solution'

28 Create a Custom Skill for Azure Cognitive Search

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

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

28.1 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/MicrosoftLe` 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**.

28.2 Create Azure resources

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

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

   ```
   az login --output none
   ```

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

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

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

    ```
    setup
    ```

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

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

- Search service query key

14. In the Azure portal, refresh the resource group and verify that it contains the Azure Storage account, Azure Cognitive Services resource, and Azure Cognitive Search resource.

28.3 Create a search solution

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

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

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

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

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

   You can find the connection string on the ***Access keys*** *page for your storage account in the Azure portal.*
3. Save and close the updated JSON file.
4. In the **create-search** folder, open **skillset.json**. This file contains a JSON definition for a skillset named **margies-custom-skillset**.
5. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_COGNITIVE_SERVIC** placeholder with either of the keys for your cognitive services resources.

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

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

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

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

28.4 Search the index

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

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

   ```
   search=London&$select=url,sentiment,keyphrases&$filter=metadata_author eq 'Reviewer' and sentiment
   ```

   This query retrieves the **url**, **sentiment**, and **keyphrases** for all documents that mention *London* authored by *Reviewer* that have a **sentiment** score greater than *0.5* (in other words, positive reviews that mention London)

28.5 Create an Azure Function for a custom skill

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

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

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

1. In Visual Studio Code, view the Azure Extensions tab ( ), and verify that the **Azure Functions** extension is installed. This extension enables you to create and deploy Azure Functions from Visual Studio Code.
2. On Azure tab (**Δ**), in the **Azure Functions** pane, create a new project ( ) with the following settings, depending on your preferred language:

28.5.1 C#

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

28.5.2 Python

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

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

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

   *If you chose Python, the code files may be in a subfolder, also named **wordcount***

4. The main code file for your function should have been opened automatically. If not, open the appropriate file for your chosen language:

   - **C#**: wordcount.cs
   - **Python**: __init__.py

5. Replace the entire contents of the file with the following code for your chosen language:

28.5.3 C#

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

namespace margies.search
{
    public static class wordcount
    {

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

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

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

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

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

            string recordId = null;
            string originalText = null;

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

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

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

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

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

                response.values.Add(responseRecord);
            }

            return (ActionResult)new OkObjectResult(response);
        }


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

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

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

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

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

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

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

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

28.6 Python

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


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

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

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

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

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

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

28.6.1 C#

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

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

28.6.2 Python

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

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

28.7 Test the function

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

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

```
{
    "values": [
        {
            "recordId": "a1",
            "data":
            {
            "text":  "Tiger, tiger burning bright in the darkness of the night.",
            "language": "en"
```

```
                }
            },
            {
                "recordId": "a2",
                "data":
                {
                "text":  "The rain in spain stays mainly in the plains! That's where you'll find the ra
                "language": "en"
                }
            }
        ]
    }
```

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

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

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

28.8 Add the custom skill to the search solution

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

1. In Visual Studio Code, in the **23-custom-search-skill/update-search** folder, open the **update-**

skillset.json file. This contains the JSON definition of a skillset.

2. Review the skillset definition. It includes the same skills as before, as well as a new **WebApiSkill** skill named **get-top-words**.
3. Edit the **get-top-words** skill definition to set the **uri** value to the URL for your Azure function (which you copied to the clipboard in the previous procedure), replacing **YOUR-FUNCTION-APP-URL**.
4. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_COGNITIVE_SERVIC** placeholder with either of the keys for your cognitive services resources.

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

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

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

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

28.9 Search the index

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

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

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

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

28.10 More information

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

28.12 lab: title: 'Create a Knowledge Store with Azure Cognitive Search' module: 'Module 12 - Creating a Knowledge Mining Solution'

29 Create a Knowledge Store with Azure Cognitive Search

Azure Cognitive Search uses an enrichment pipeline of cognitive skills to extract AI-generated fields from documents and include them in a search index. While the index might be considered the primary output from an indexing process, the enriched data it contains might also be useful in other ways. For example:

- Since the index is essentially a collection of JSON objects, each representing an indexed record, it might be useful to export the objects as JSON files for integration into a data orchestration process using tools such as Azure Data Factory.
- You may want to normalize the index records into a relational schema of tables for analysis and reporting with tools such as Microsoft Power BI.
- Having extracted embedded images from documents during the indexing process, you might want to save those images as files.

In this exercise, you'll implement a knowledge store for *Margie's Travel*, a fictitious travel agency that uses information in brochures and hotel reviews to help customers plan trips.

29.1 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/MicrosoftLe` 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**.

29.2 Create Azure resources

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

1. In a web browser, open the Azure portal at `https://portal.azure.com`, and sign in using the Microsoft account associated with your Azure subscription.

2. View the **Resource groups** in your subscription.

3. If you are using a restricted subscription in which a resource group has been provided for you, select the resource group to view its properties. Otherwise, create a new resource group with a name of your choice, and go to it when it has been created.

4. On the **Overview** page for your resource group, note the **Subscription ID** and **Location**. You will need these values, along with the name of the resource group in subsequent steps.

5. In Visual Studio Code, expand the **24-knowledge-store** folder and select **setup.cmd**. You will use this batch script to run the Azure command line interface (CLI) commands required to create the Azure resources you need.

6. Right-click the the **24-knowledge-store** folder and select **Open in Integrated Terminal**.

7. In the terminal pane, enter the following command to establish an authenticated connection to your Azure subscription.

```
az login --output none
```

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

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

10. In the output, find the **Name** value that corresponds with the location of your resource group (for example, for *East US* the corresponding name is *eastus*).
11. In the **setup.cmd** script, modify the **subscription_id**, **resource_group**, and **location** variable declarations with the appropriate values for your subscription ID, resource group name, and location name. Then save your changes.
12. In the terminal for the **24-knowledge-store** folder, enter the following command to run the script:

   ```
   setup
   ```

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

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

29.3 Create a search solution

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

- A **data source** that references the documents in your Azure storage container.
- A **skillset** that defines an enrichment pipeline of skills to extract AI-generated fields from the documents. The skillset also defines the *projections* that will be generated in your *knowledge store*.
- An **index** that defines a searchable set of document records.
- An **indexer** that extracts the documents from the data source, applies the skillset, and populates the index. The process of indexing also persists the projections defined in the skillset in the knowledge store.

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

29.3.1 Prepare JSON for REST operations

You'll use the REST interface to submit JSON definitions for your Azure Cognitive Search components.

1. In Visual Studio Code, in the **24-knowledge-store** folder, expand the **create-search** folder and select **data_source.json**. This file contains a JSON definition for a data source named **margies-knowledge-data**.
2. Replace the **YOUR_CONNECTION_STRING** placeholder with the connection string for your Azure storage account, which should resemble the following:

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

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

3. Save and close the updated JSON file.

4. In the **create-search** folder, open **skillset.json**. This file contains a JSON definition for a skillset named **margies-knowledge-skillset**.

5. At the top of the skillset definition, in the **cognitiveServices** element, replace the **YOUR_COGNITIVE_SERVIC** placeholder with either of the keys for your cognitive services resources.

   *You can find the keys on the **Keys and Endpoint** page for your cognitive services resource in the Azure portal.*

6. At the end of the collection of skills in your skillset, find the **Microsoft.Skills.Util.ShaperSkill** skill named **define-projection**. This skill defines a JSON structure for the enriched data that will be used for the projections that the pipeline will persist on the knowledge store for each document processed by the indexer.

7. At the bottom of the skillset file, observe that the skillset also includes a **knowledgeStore** definition, which includes a connection string for the Azure Storage account where the knowledge store is to be created, and a collection of **projections**. This skillset includes three *projection groups*:

   - A group containing an *object* projection based on the **knowledge_projection** output of the shaper skill in the skillset.
   - A group containing a *file* projection based on the **normalized_images** collection of image data extracted from the documents.
   - A group containing the following *table* projections:
     - **KeyPhrases**: Contains an automatically generated key column and a **keyPhrase** column mapped to the **knowledge_projection/key_phrases/** collection output of the shaper skill.
     - **Locations**: Contains an automatically generated key column and a **location** column mapped to the **knowledge_projection/key_phrases/** collection output of the shaper skill.
     - **ImageTags**: Contains an automatically generated key column and a **tag** column mapped to the **knowledge_projection/image_tags/** collection output of the shaper skill.
     - **Docs**: Contains an automatically generated key column and all of the **knowledge_projection** output values from the shaper skill that are not already assigned to a table.

8. Replace the **YOUR_CONNECTION_STRING** placeholder for the **storageConnectionString** value with the connection string for your storage account.

9. Save and close the updated JSON file.

10. In the **create-search** folder, open **index.json**. This file contains a JSON definition for an index named **margies-knowledge-index**.

11. Review the JSON for the index, then close the file without making any changes.

12. In the **create-search** folder, open **indexer.json**. This file contains a JSON definition for an indexer named **margies-knowledge-indexer**.

13. Review the JSON for the indexer, then close the file without making any changes.

29.3.2 Submit REST requests

Now that you've prepared the JSON objects that define your search solution components, you can submit the JSON documents to the REST interface to create them.

1. In the **create-search** folder, open **create-search.cmd**. This batch script uses the cURL utility to submit the JSON definitions to the REST interface for your Azure Cognitive Search resource.

2. Replace the **YOUR_SEARCH_URL** and **YOUR_ADMIN_KEY** variable placeholders with the **Url** and one of the **admin keys** for your Azure Cognitive Search resource.

   *You can find these values on the **Overview** and **Keys** pages for your Azure Cognitive Search resource in the Azure portal.*

3. Save the updated batch file.

4. Right-click the the **create-search** folder and select **Open in Integrated Terminal**.

5. In the terminal pane for the **create-search** folder, enter the following command run the batch script.

   ```
   create-search
   ```

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

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

   > **Tip**: If the script fails, check the placeholders you added in the **data_source.json** and **skillset.json** files as well as the **create-search.cmd** file. After correcting any mistakes, you may need to use the Azure portal user interface to delete any components that were created in your search resource before re-running the script.

29.4 View the knowledge store

After you have run an indexer that uses a skillset to create a knowledge store, the enriched data extracted by the indexing process is persisted in the knowledge store projections.

29.4.1 View object projections

The *object* projections defined in the Margie's Travel skillset consist of a JSON file for each indexed document. These files are stored in a blob container in the Azure Storage account specified in the skillset definition.

1. In the Azure portal, view the Azure Storage account you created previously.
2. Select the **Storage explorer** tab (in the pane on the left) to view the storage account in the storage explorer interface in the Azure portal.
3. Expand **BLOB CONTAINERS** to view the containers in the storage account. In addition to the **margies** container where the source data is stored, there should be two new containers: **margies-images** and **margies-knowledge**. These were created by the indexing process.
4. Select the **margies-knowledge** container. It should contain a folder for each indexed document.
5. Open any of the folders, and then download and open the **knowledge-projection.json** file it contains. Each JSON file contains a representation of an indexed document, including the enriched data extracted by the skillset as shown here.

```
{
    "file_id":"abcd1234....",
    "file_name":"Margies Travel Company Info.pdf",
    "url":"https://store....blob.core.windows.net/margies/...pdf",
    "language":"en",
    "sentiment":0.83164644241333008,
    "key_phrases":[
        "Margie’s Travel",
        "Margie's Travel",
        "best travel experts",
        "world-leading travel agency",
        "international reach"
        ],
    "locations":[
        "Dubai",
        "Las Vegas",
        "London",
        "New York",
        "San Francisco"
        ],
    "image_tags":[
        "outdoor",
        "tree",
        "plant",
        "palm"
        ]
}
```

The ability to create *object* projections like this enables you to generate enriched data objects that can be incorporated into an enterprise data analysis solution - for example by ingesting the JSON files into an Azure Data Factory pipeline for further processing or loading into a data warehouse.

29.4.2 View file projections

The *file* projections defined in the skillset create JPEG files for each image that was extracted from the documents during the indexing process.

1. In the storage explorer interface in the Azure portal, select the **margies-images** blob container. This container contains a folder for each document that contained images.
2. Open any of the folders and view its contents - each folder contains at least one *.jpg file.
3. Open any of the image files to verify that they contain images extracted from the documents.

The ability to generate *file* projections like this makes indexing an efficient way to extract embedded images from a large volume of documents.

29.4.3 View table projections

The *table* projections defined in the skillset form a relational schema of enriched data.

1. In the storage explorer interface in the Azure portal, expand **TABLES**.
2. Select the **Docs** table to view its columns. The columns include some standard Azure Storage table columns - to hide these, modify the **Column Options** to select only the following columns:
    - **document_id** (the key column automatically generated by the indexing process)
    - **file_id** (the encoded file URL)
    - **file_name** (the file name extracted from the document metadata)
    - **language** (the language in which the document is written)
    - **sentiment** the sentiment score calculated for the document.
    - **url** the URL for the document blob in Azure storage.
3. View the other tables that were created by the indexing process:
    - **ImageTags** (contains a row for each individual image tag with the **document_id** for the document in which the tag appears).
    - **KeyPhrases** (contains a row for each individual key phrase with the **document_id** for the document in which the phrase appears).
    - **Locations** (contains a row for each individual location with the **document_id** for the document in which the location appears).

The ability to create *table* projections enables you to build analytical and reporting solutions that query the relational schema; for example, using Microsoft Power BI. The automatically generated key columns can be used to join the tables in queries - for example to return all of the locations mentioned in a specific document.

29.5 More information

To learn more about creating knowledge stores with Azure Cognitive Search, see the Azure Cognitive Search documentation.