Download https://github.com/szelor/Hekaton repository to your local machine and unpack it to c:\hekaton folder.

# Lab 0 Cognitive Services

## Try out the Cognitive Services using the website demo options

Navigate to: https://azure.microsoft.com/en-gb/services/cognitive-services/directory/vision/. There are lots of different demos to try in each section (Scene and Activity Recognition in Images, OCR, Face Detection, Emotion Detection, Video indexer etc).

Select the Demo link next to Scene and activity recognition in images under Computer Vision. There are also other demo links to explore the different services.

Now select Browse button and upload the cat.jpeg or city.jpeg image from sample-images/computer-vision-web-browser/.

Set up your Azure Account

You may activate an Azure free trial at https://azure.microsoft.com/en-us/free/.

If you have been given an Azure Pass to complete this lab, you may go to http://www.microsoftazurepass.com/ to activate it. Please follow the instructions at https://www.microsoftazurepass.com/howto, which document the activation process. A Microsoft account may have one free trial on Azure and one Azure Pass associated with it, so if you have already activated an Azure Pass on your Microsoft account, you will need to use the free trial or use another Microsoft account.

## Provision Cognitive Services keys

First log into Microsoft Azure and choose Portal in the top right corner. Once in the portal select Create a resource and search Cognitive Services and choose Enter. Then select Create on the Cognitive Services blade.

Enter details to create an account:

* Name: enter a suitable name for the service (example: hekatoncognitive)
* Subscription: Choose your subscription
* Location: Choose West US
* Pricing Tier: S0
* Resource Group: Select 'Create new', and provide a sensible name (example hekaton)
* select the checkbox after reading the terms below
* select 'Create'

Once created, in your notifications (top right corner) select go to resource.

In the Cognitive Services page, select Keys and copy KEY 1. Save it in a text file so you can easily access it in future labs.

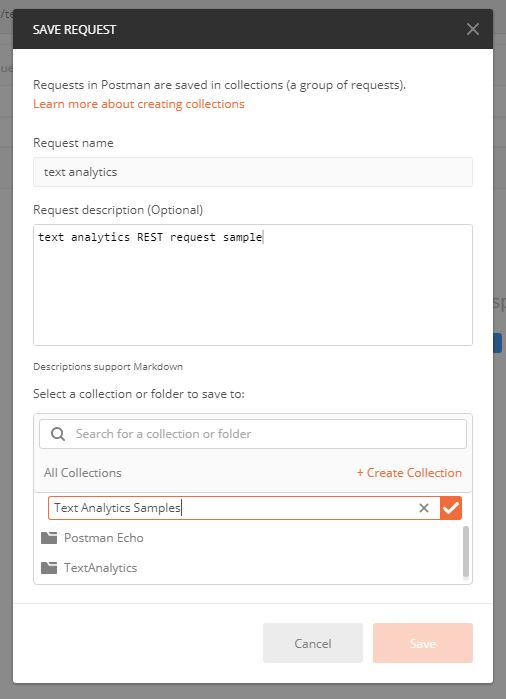
Now select Overview in the left hand pane and copy the Endpoint variable.

## Text Analytics via REST

Download (https://www.getpostman.com/downloads/) and open Postman, an API Development environment on your local machine.

Select Request

Enter request details as below and choose the option create a new collection and name it "Text Analytics Samples"



Select the newly created collection and choose save

Now create a request to call your text analytics API:

* Change from a GET request to a POST request in the top left
* Enter your endpoint URL and add text/analytics/v2.0/sentiment to the end
* Select Headers underneath the URL box
* In Key type Ocp-Apim-Subscription-Key and in Value add your KEY1 value
* In Key type Content-Type and in Value type application/json
* Select Body underneath the URL box
* Select raw from the radio button options
* Copy JSON sample from sample-code/cognitive-services-api-task/sentiment-analysis-text.json into the box
* Select the Send button and review the Response

You can also try other options from the REST API - such as KeyPhrases function. Change the end of the URL from sentiment to keyPhrases and select send to view the key phrases for the example text.

Check out the language support for the Text Analytics API. If your language is supported please edit the JSON file to translate the text and show the functionality of the API above. There is an example of a French JSON file in sample-code/text-analytics-demo/sentiment-analysis-text-fr.json.

# Lab 1 Computer Vision

## Set up Storage

We'll be using two different stores in Azure for this project - one for storing the raw images, and the other for storing the results of our Cognitive Service calls. Azure Blob Storage is made for storing large amounts of data in a format that looks similar to a file-system, and it is a great choice for storing data like images. Azure Cosmos DB is our resilient NoSQL PaaS solution and is incredibly useful for storing loosely structured data like we have with our image metadata results. There are other possible choices (Azure Table Storage, SQL Server), but Cosmos DB gives us the flexibility to evolve our schema freely (like adding data for new services), query it easily, and integrate quickly into Azure Search.

### Azure Blob Storage

In the Portal, click the "+ New" button (when you hover over it, it will say Create a resource) and then enter storage in the search box and choose Storage account. Select create.

Once you click it, you'll be presented with some fields to fill out.

* Choose your storage account name (lowercase letters and numbers),
* Set Account kind to Blob storage,
* Set Replication to Locally-Redundant storage (LRS) (this is just to save money),
* Use the same Resource Group as above, and
* Set Location to the region that is closest to you from the following list: East US, West US, Southeast Asia, West Europe
* All other defaults are fine.

### Cosmos DB

In the Portal, click the "+ New" button (when you hover over it, it will say Create a resource) and then enter cosmos db in the search box and choose Azure Cosmos DB and click Create.

Once you click this, you'll have to fill out a few fields as you see fit. Set Location to the region that is closest to you from the following list: East US, West US, Southeast Asia, West Europe.

In our case, select the ID you'd like, subject to the constraints that it needs to be lowercase letters, numbers, or dashes. We will be using the SQL API so we can create a document database that is queryable using SQL syntax, so select SQL as the API. Let's use the same Resource Group as we used for our previous steps, and the same location, select Pin to dashboard to make sure we keep track of it and it's easy to get back to, and hit Create.

## Collect the Keys

Over the course of this lab, we will collect Cognitive Services keys and storage keys. You should save all of them in a text file so you can easily access them in future labs.

Cognitive Services Keys

Computer Vision API:

Storage Keys

Azure Blob Storage Connection String:

Cosmos DB URI:

Cosmos DB key:

In addition, you will need to add the keys to the settings.json file which is under C:\Hekaton\lab1-computer\_vision\resources\code\Starting-ImageProcessing\TestCLI\settings.json. You will have to replace VisionKeyHere, ConnectionStringHere, CosmosURIHere, and CosmosKeyHere with their corresponding keys that you collect in the next section. Do not change the blob container (images), the database name (images), or the collection name (metadata).

## Image Processing

Within your solution, under resources>code>Starting-ImageProcessing, you'll find the Processing Library. This is a Portable Class Library (PCL), which helps in building cross-platform apps and libraries quickly and easily. It serves as a wrapper around several services. This specific PCL contains some helper classes (in the ServiceHelpers folder) for accessing the Computer Vision API and an "ImageInsights" class to encapsulate the results. Later, we'll create an image processor class that will be responsible for wrapping an image and exposing several methods and properties that act as a bridge to the Cognitive Services. After creating the image processor, you should be able to pick up this portable class library and drop it in your other projects that involve Cognitive Services.

### ProcessingLibrary: Service Helpers

Service helpers can be used to make your life easier when you're developing your app. One of the key things that service helpers do is provide the ability to detect when the API calls return a call-rate-exceeded error and automatically retry the call (after some delay). They also help with bringing in methods, handling exceptions, and handling the keys.

You can find additional service helpers for some of the other Cognitive Services within the Intelligent Kiosk sample application. Utilizing these resources makes it easy to add and remove the service helpers in your future projects as needed.

### ProcessingLibrary: The "ImageInsights" class

Take a look at the "ImageInsights" class. You can see that we're calling for Caption and Tags from the images, as well as a unique ImageId. "ImageInsights" pieces only the information we want together from the Computer Vision API (or from Cognitive Services, if we choose to call multiple).

Now let's take a step back for a minute. It isn't quite as simple as creating the "ImageInsights" class and copying over some methods/error handling from service helpers. We still have to call the API and process the images somewhere. For the purpose of this lab, we are going to walk through creating ImageProcessor.cs, but in future projects, feel free to add this class to your PCL and start from there (it will need modification depending what Cognitive Services you are calling and what you are processing - images, text, voice, etc.).

### Creating ImageProcessor.cs

Right-click on the solution and select "Build Solution". If you have errors related to ImageProcessor.cs, you can ignore them for now, because we are about to address them.

* Navigate to ImageProcessor.cs within ProcessingLibrary
* Add the following using directives to the top of the class, above the namespace

using System;

using System.IO;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.ProjectOxford.Vision;

using ServiceHelpers;

* In ImageProcessor.cs we will start by creating a method we will use to process the image, ProcessImageAsync. Paste the following code within the ImageProcessor class (between the { })

public static async Task<ImageInsights> ProcessImageAsync(Func<Task<Stream>> imageStreamCallback, string imageId)

{

* In the above code, we use Func<Task<Stream>> because we want to make sure we can process the image multiple times (once for each service that needs it), so we have a Func that can hand us back a way to get the stream. Since getting a stream is usually an async operation, rather than the Func handing back the stream itself, it hands back a task that allows us to do so in an async fashion.
* In ImageProcessor.cs, within the ProcessImageAsync method, we're going to set up a static array that we'll fill in throughout the processor. As you can see, these are the main attributes we want to call for ImageInsights.cs. Add the code below between the { } of ProcessImageAsync:

VisualFeature[] DefaultVisualFeaturesList = new VisualFeature[] { VisualFeature.Tags, VisualFeature.Description };

* Next, we want to call the Cognitive Service (specifically Computer Vision) and put the results in imageAnalysisResult. Use the code below to call the Computer Vision API (with the help of VisionServiceHelper.cs) and store the results in imageAnalysisResult. Near the bottom of VisionServiceHelper.cs, you will want to review the available methods for you to call (RunTaskWithAutoRetryOnQuotaLimitExceededError, DescribeAsync, AnalyzeImageAsync, RecognizeTextAsyncYou). Replace the "\_" with the method you need to call in order to return the visual features.

var imageAnalysisResult = await VisionServiceHelper.\_(imageStreamCallback, DefaultVisualFeaturesList);

* Now that we've called the Computer Vision service, we want to create an entry in "ImageInsights" with only the following results: ImageId, Caption, and Tags (you can confirm this by revisiting ImageInsights.cs). Paste the following code below var imageAnalysisResult and try to fill in the code for ImageId, Caption, and Tags:

ImageInsights result = new ImageInsights

{

ImageId = ,

Caption = ,

Tags =

};

Depending on your C# dev background, this may not be an easy task. Here are some hints:

* When you defined the method, you specified a result. This should help you determine what ImageId is (it's actually really simple).
* For Caption and Tags, start with imageAnalysisResult.. When you put the ., you're able to use the dropdown menu to help.
* For Caption, you have to specify that you want only the first caption by using [0].
* For Tags, you have to put the tags into an array with Select(t => t.Name).ToArray().
* Still stuck? You can take a peek at the solution at resources>code>Finished-ImageProcessing>ProcessingLibrary>ImageProcessor.cs

So now we have the caption and tags that we need from the Computer Vision API, and each image's result (with imageId) is stored in "ImageInsights".

* Lastly, we need to close out the method by adding the following line to the end of the method:

return result;

Now that you've built ImageProcessor.cs, don't forget to save it!

Loading Images using TestCLI

We will implement the main processing and storage code as a command-line/console application because this allows you to concentrate on the processing code without having to worry about event loops, forms, or any other UX related distractions. Feel free to add your own UX later.

Once you've set your Cognitive Services API keys, your Azure Blob Storage Connection String, and your Cosmos DB Endpoint URI and Key in your TestCLI's settings.json, you can run the TestCLI.

Run TestCLI, then open Command Prompt and navigate to "C:\Hekaton\Inteligent-Search\lab1-computer\_vision\resources\code\Starting-ImageProcessing\TestCLI\bin\Debug" folder (Hint: use the "cd" command to change directories). Then enter TestCLI.exe. You should get the following result:

> TestCLI.exe

Usage: [options]

Options:

-force Use to force update even if file has already been added.

-settings The settings file (optional, will use embedded resource settings.json if not set)

-process The directory to process

-query The query to run

-? | -h | --help Show help information

By default, it will load your settings from settings.json (it builds it into the .exe), but you can provide your own using the -settings flag. To load images (and their metadata from Cognitive Services) into your cloud storage, you can just tell TestCLI to -process your image directory as follows:

TestCLI.exe -process C:\Hekaton\Inteligent-Search\lab1-computer\_vision\resources\sample\_images

Once it's done processing, you can query against your Cosmos DB directly using TestCLI as follows:

TestCLI.exe -query "select \* from images"

# Lab 2A Custom Vision and Logic Apps

Using Microsoft Azure Custom Vision service you can start to build your own personalized image classification and object detection algorithms with very little code. In this exercise we will create a dog-breed classification algorithm using Dog images from the ImageNet open dataset created by Standford University.

We have 7 Classes of dogs each with 30 images available in \Cognitive Services\sample-images\dogs folder

* Beagle
* Bernese Mountain Dog
* Chihuahua
* Eskimo Dog (aka Husky)
* German Shepherd
* Golden Retriever
* Maltese

There is also a set of test images (not for training) in this folder.

## Resource provisioning

First create a Custom Vision instance in your Azure account.

* Go to the Azure Portal main dashboard.
* Click 'Create a Resource' in the top left
* Search for 'Custom Vision'
* On the description pane for Custom Vision click Create.
* Enter details to create
  + a name for the service
  + select your subscription
  + Please choose West US as the data centre
  + Choose the S0 tier for both 'Prediction pricing tier' and Training pricing tier
  + select the resource group you created previously for this project (e.g. ainights)
  + Click Create

## Create Custom Vision project

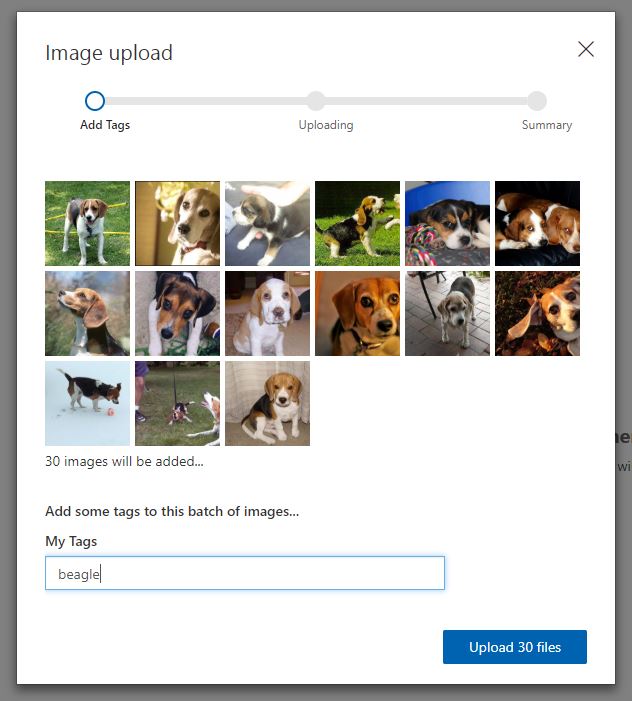
Now we can build our classifier, navigate to https://www.customvision.ai and choose sign in. Sign in with your Azure credentials account.

Once loaded choose 'New Project' which opens a window to enter details

* Name: choose a suitable name
* Description: add a description of the classifier (example shown in image below)
* Resource Group: choose the resource group you created your custom vision service in (example: hekaton[SO])
* Project Types: Classification
* Classification Types: Multiclass (Single tag per image)
* Domains: General

Choose 'Create Project' and you will land on an empty workspace. Now we can start adding images and assigning them tags to create our image classifier.

In the top left, select 'Add images', browse for the first folder of images from the Dogs folder - Beagle - and select all 30 of the images in the folder. Add the tag 'beagle' to the Beagle dog images and select 'Upload 30 files'.



Once successful you receive a confirmation message and you should see your images are now available in the workspace.

Now complete the same steps of uploading and tagging images for the other 6 dog categories in the folder. For each type of dog:

* Click add images
* Select the 30 new dog images
* Add the class label (beagle, german-shepherd, maltese etc)
* choose upload
* confirm images uploaded into the workspace

Now you should have all categories uploaded and on the left hand side you can see your dog classes and you can filter depending on type of dog image.

Now you are ready to train your algorithm on the dog image data you have uploaded. Select the green 'Train' button in the top right corner.

Once the training process is complete it will take you to the Performance tab. Here you will receive machine learning evaluation metrics for your model.

Now we have a model we need to test the model. Choose the 'Quick Test' button in the top right (next to the train button) this will open a window where you can browse for a local image or enter a web URL.

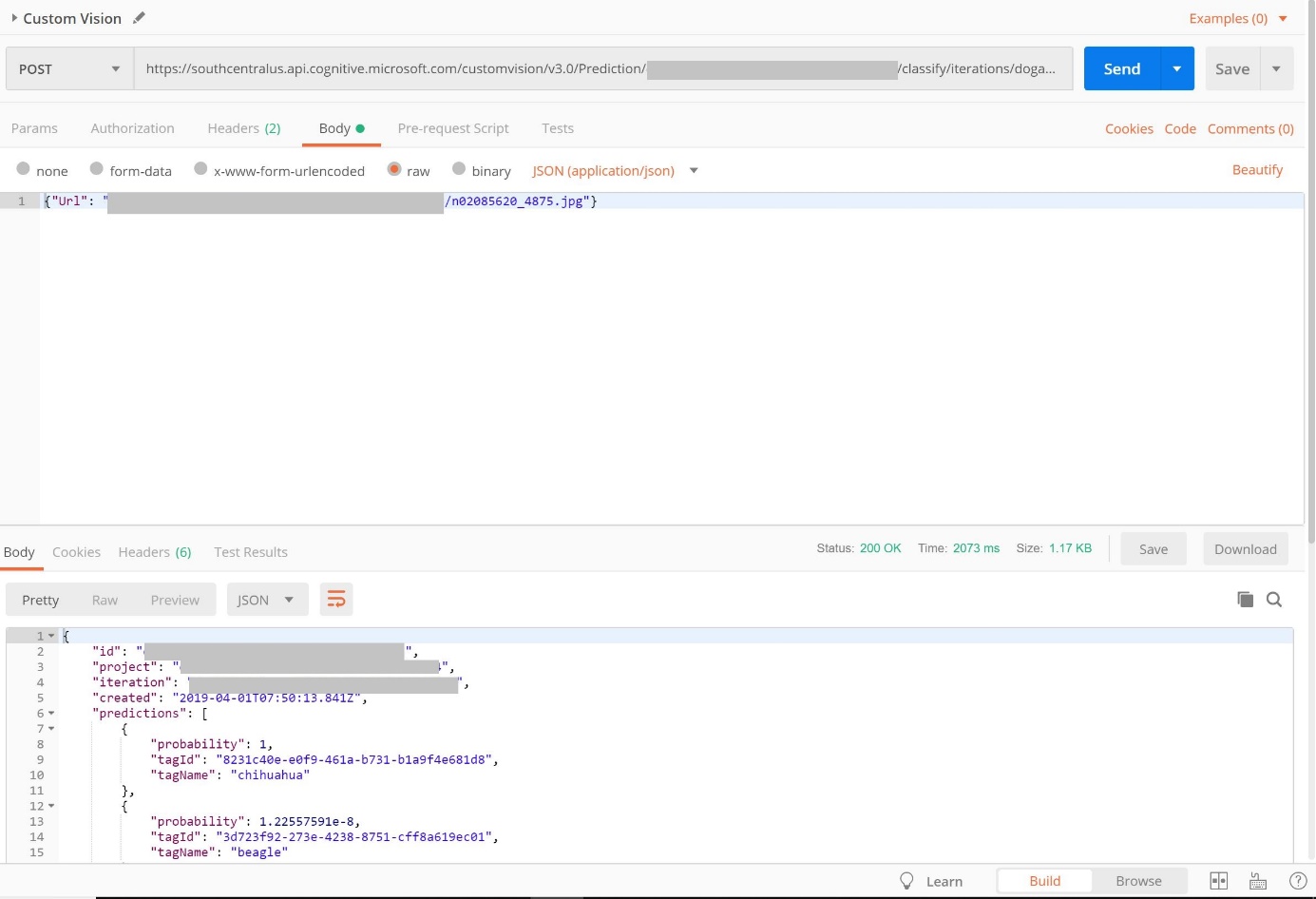
Browse for an image in the test folder (images the model have not been trained on) and upload. The image will be analysed and a result returned of what dog the model thinks it is (prediction tag) and the models confidence of its result (prediction probability).

If you click on the 'Predictions' tab on the top toolbar - you should see all the test images you have submitted. This section is for re-training, as you get new data you can add this to your model to improve its performance. The images are ordered by importance - the image, which if classified correctly, will add the most new information to the model is listed first. Whereas the last image might be very similar to other images already learnt by the model so this is less important to classify correctly. To add these images to the model - select the first image, review the results the model provided and then in the 'My Tags' box enter the correct tag and click 'save and close'.

This image will disappear from the your predictions workspace and be added to the training images workspace. Once you add a few new images and tags you can re-train the model to see if there are improvements.

To use this model within applications you need the prediction details. Therefore, you have to go to the Performance tab from the top bar, click the Publish button and provide a name for this published iteration.

You can now select the Prediction URL button to gain all information you need to create a Postman call to your API, by setting the URL, the Header and the Body (using both an image or an image URL)



Great work! you have created your specialised dog classification model using the Azure Custom Vision Service

## Build Custom AI into an Application using Azure Logic Apps

In this section you will build an Azure Logic App to consume your Custom Vision AI dog classification application

First we need to create two Azure Storage Accounts.

Go to the azure portal and click create new resource in the top left corner. Select the section Storage and choose the first option Storage Account. We are going to create two storage accounts:

* one for the images to be dropped into to be processed (called storagea)
* another for the results after processing to be uploaded to (called storageb)

Complete the process below twice so you have two storage accounts in total

On the storage account creation page enter options to setup your storage account:s

* Subscription: choose your subscription
* Resource Group: choose the resource group you have been using for this workshop (e.g. ainights)
* Storage Account Name: (must be unique) enter an all lowercase storage account name. Such as ainightsstor(yourname) or resultsainights(yourname) - append your name to the end of the storage account name so you know its unique (remove the brackets)
* Location: your closest data center
* Performance: Standard
* Account Kind: Blob Storage
* Replication: Locally-redundant storage (LRS)
* Access Tier: Hot
* Select Review + create, confirm validation is passed and then select Create.

Once your deployment is complete, got to the resource and review the account settings. Select Blobs to review your empty blob storage account.

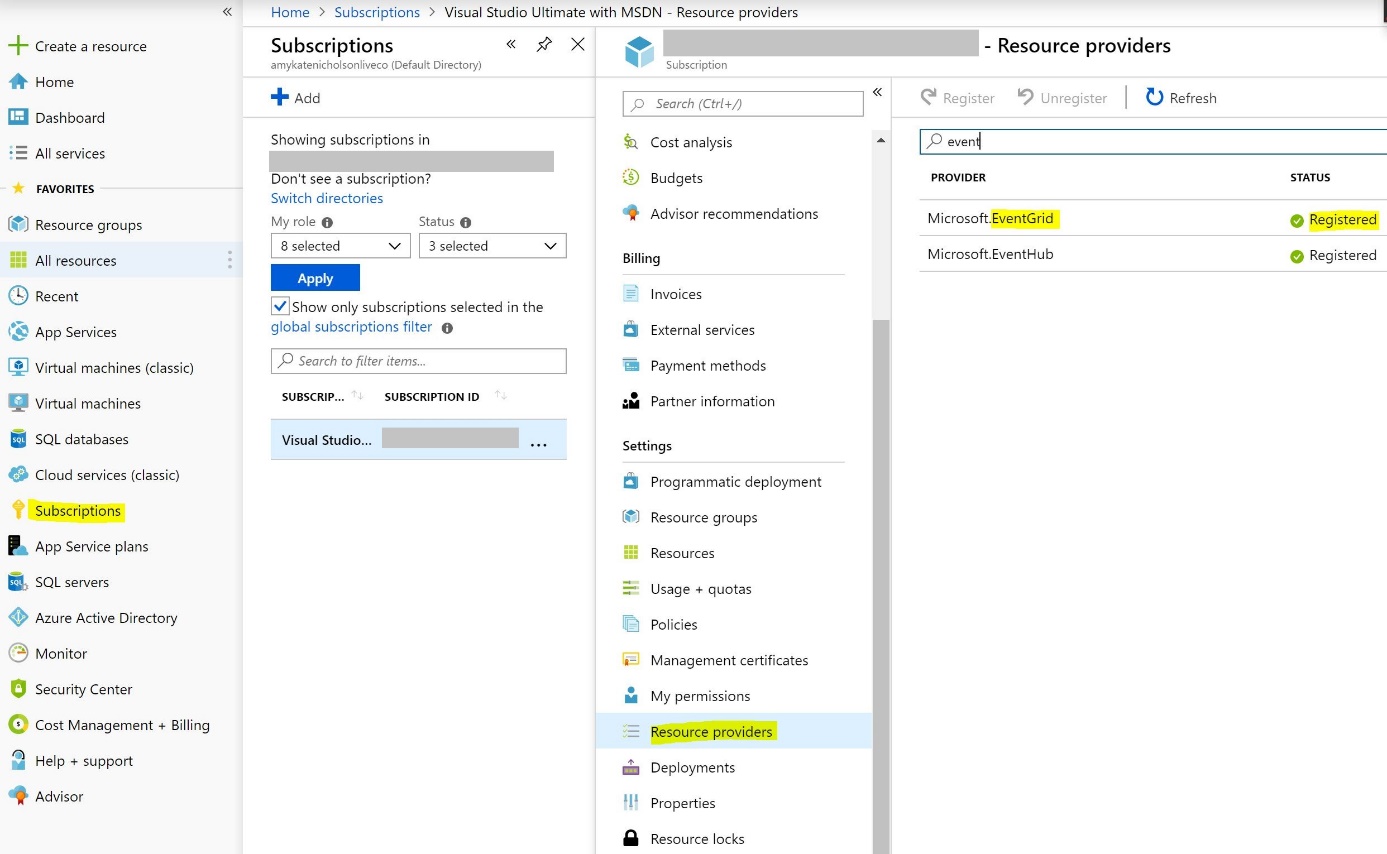
We need to add a container to the storage account to store our images and results.

* Select the + Container button and create a name for the container
* an example for the ainightsstor account would be images
* an example for the resultsainights account would be results
* For the public access level setting select Container (anonymous read access for containers and blobs)

Now we will create a Logic App - this will connect your image storage account to your AI classification service and put the results in your results storage account.

Head to the Azure Portal Homepage. We are going to use Event Grid, a service that detects triggers in an Azure subscription (in our case, when a new blob is created in your Azure Storage account). Before we build with this - we must register it.

Got to subscriptions in the left panel, select your subscription and find Resource Providers in the left pane. Once the resource providers are listed - search "event" and select Microsoft.EventGrid.



If this is not already status registered, select register from the toolbar

Once registered with a green tick - go back to the Azure Portal Homepage. Select Create a Resource. Type Logic App and select the service

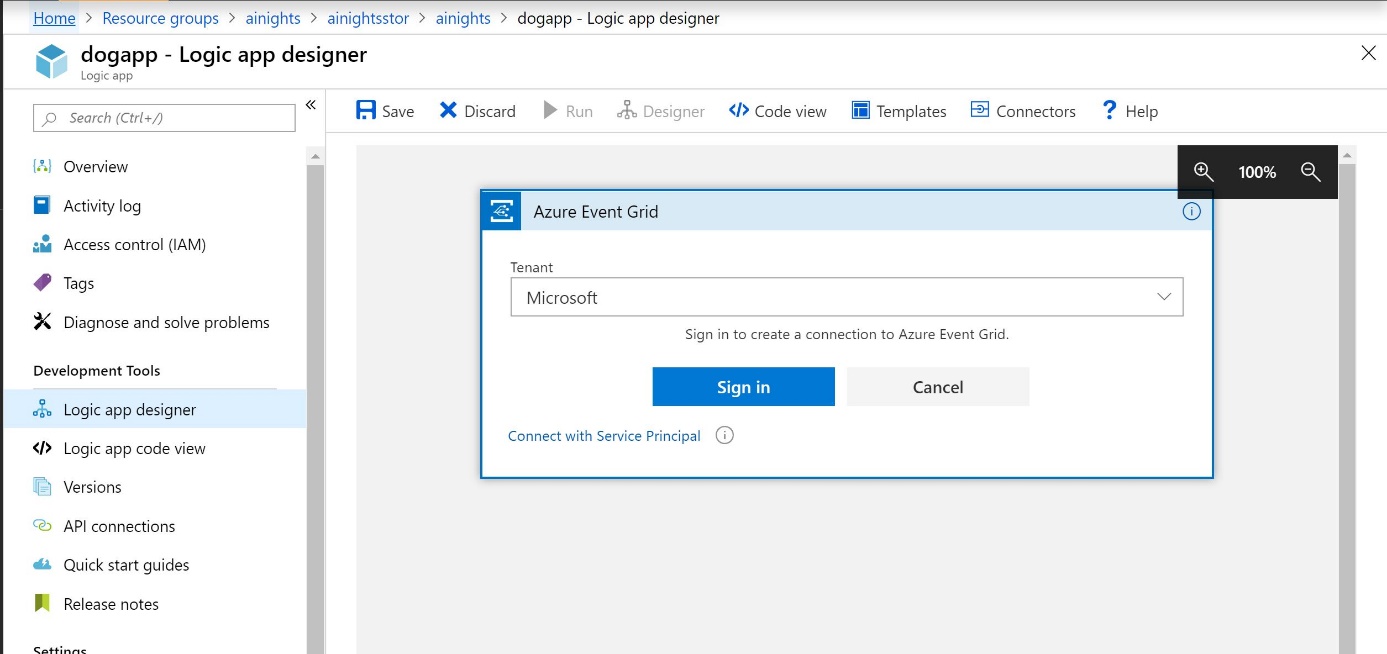
Create the logic app by entering some setup detail like below:

* Name: suitable name for the dog classification application
* Subscription: Choose your subscription
* Resource Group: (use existing e.g. ainights) select the resource group you have been working for the whole workshop
* Location: choose the data center closest to you
* Log Analytics: off

Choose Create

Once created, go to resource. From here we can create our logic process. Select Logic app designer from the left menu and then the When an Event Grid resource event occurs option.

Connect to Azure event grid by signing in using your Azure credentials



Once connected and you see a green tick, select continue.

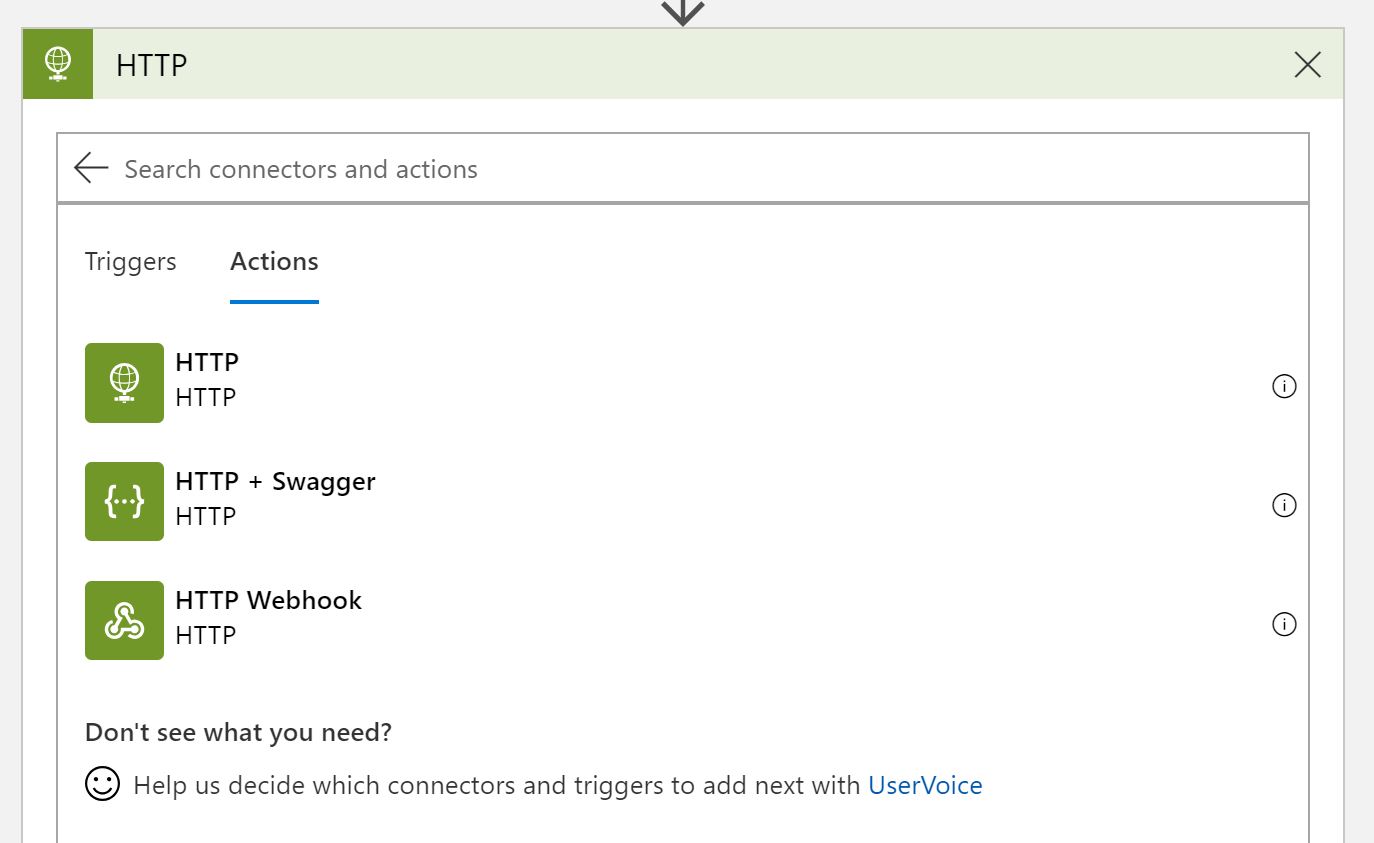
Select the options below:

* Subscription: your subscription
* Resource Type: Microsoft.Storage.StorageAccounts
* Resource Name: choose your image storage account (e.g. ainightsstor)
* Event Type Item - 1: Microsoft.Storage.BlobCreated

Then choose next step. Type Parse JSON and select the parse JSON operator as part of the data Data Operations category

* Content: select the box and from the Dynamic Content box on the right, select Body
* Schema: select this box and enter the JSON schema provided in the logic-app-schema1 file

Then choose next step. Type HTTP and select the HTTP option as below



Now we need to fill in the details of the REST API request - similar to using Postman App.

* Method: POST
* URI: enter Prediction URL from Custom Vision Service
* Headers:
  + "Prediction-Key" : enter your prediction key from the custom vision service
  + "Content-Type" : "application/json"
* Queries: enter nothing
* Body: {"Url": "REPLACE WITH DYNAMIC CONTENT URL"}

Then choose next step. Type Parse JSON and select the parse JSON operator again as part of the data Data Operations category

* Content: select the box and from the Dynamic Content box on the right, select Body
* Schema: select this box and enter the JSON schema provided in the logic-app-schema2 file

Choose next step.

Type for each and select the grey control step called for each Once selected in the output from previous step box, select the box and from Dynamic content select predictions from the Parse JSON 2 category.

Choose Add an action.

Search Control, select the control icon and then from the results, select Condition

In the If True box select Add an action

Search for Azure Blob Storage and select the icon for Create Blob

In connection name enter results and select your results blob storage account name from the listed options and select create

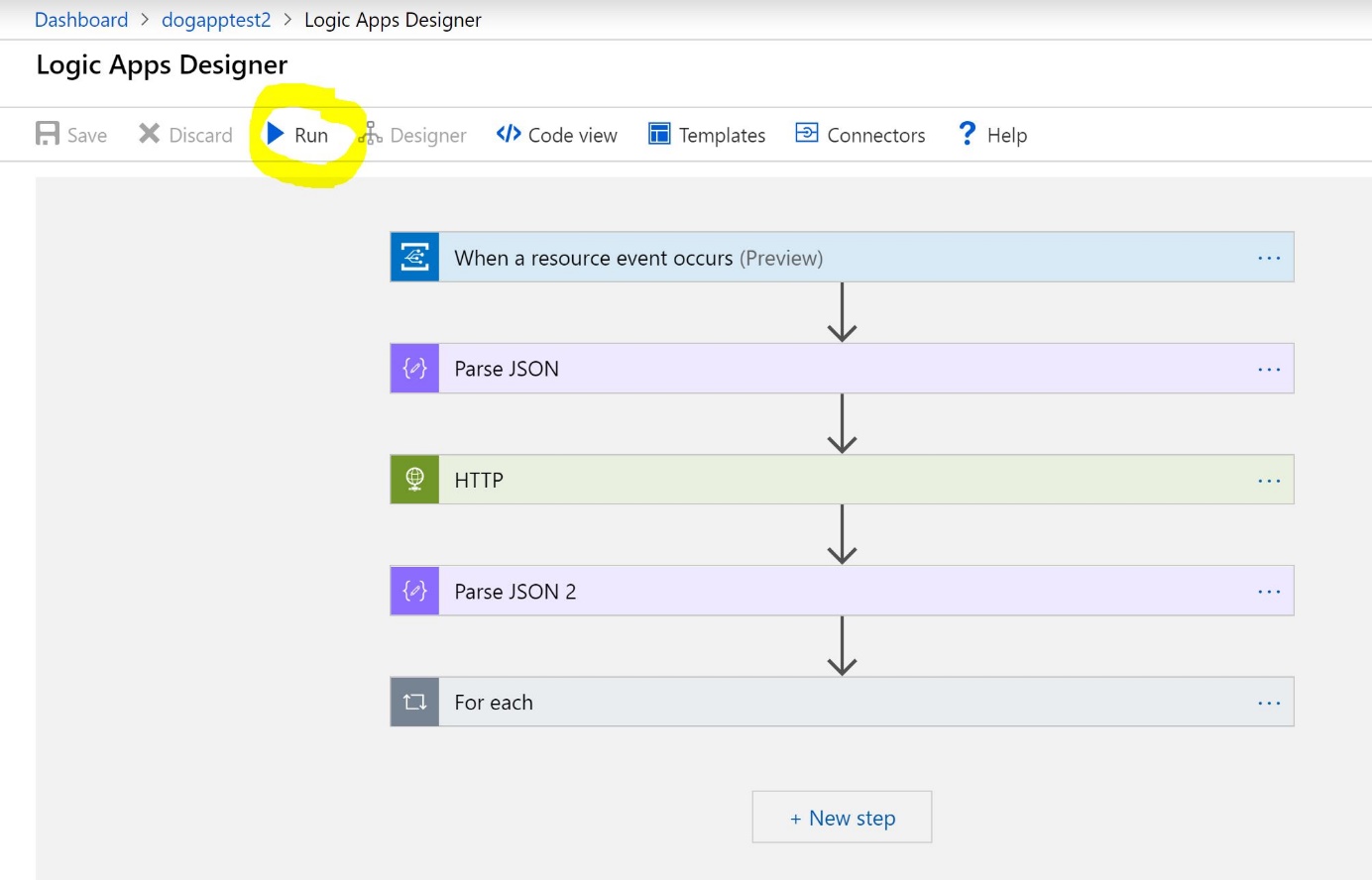
In folder path, select the folder icon, far right, and choose the container name you created that is populated

Select the Blob name field and enter: result-(then from the Dynamic content box under Parse Json (1) select id)

Under Blob Content, select the field and in the Dynamic Content box on the right, select see more under the Parse Json 2 section. Then select tagName, enter a colon ":" and then select probability.

Finally save the logic app in the top action bar

Once saved, lets test the app for the desired outcome. Select Run from the top action bar



Now navigate to your images storage account (easy to find from the resource group section). Choose Blob and select the images container. In there you should see an upload button. Upload one of the images from the Dogs data testset folder.

Once uploaded, navigate back to your Logic App main page and review the runs history. All sections should have a green tick and you can select each one to view the input and output between the layers (this is also a great way to debug if it doesn't run as expected).

Finally navigate to your results blob storage account, select blob, enter the results container and review the file now created there.

# Lab 2B Creating an Object Detection Application using the Custom Vision API

The goal of this tutorial is to explore a basic Windows application that uses the Custom Vision API to create an object detection project, add tags to it, upload images, train the project, obtain the default prediction endpoint URL for the project, and use the endpoint to programmatically test an image. You can use this open source example as a template for building your own app for Windows using the Custom Vision API.

Long files name can cause Nuget package failures. Should you receive this error, it is recommended that you place the solution files in a folder you have created to reduce the number of characters in the filepath. If you receive the following error „NuGet Package restore failed for project ObjectDetection: The specified path, file name, or both are too long. The fully qualified file name must be less than 260 characters, and the directory name must be less than 248 characters. Please see Error List window for detailed warnings and errors.” perform the following steps

* Create a folder in the C:\ named Temp
* In Windows Explorer, browse to the folder C:\Hekaton\Intelligent-Search\lab2\_customvision\
* Copy the Resources folder
* Browse to C:\Temp, right click and click Paste
* Wait until the copy completes

## Prerequisites

You need to have a training API key. The training API key allows you to create, manage, and train Custom Vision projects programmatically. All operations on https://customvision.ai are exposed through this library, allowing you to automate all aspects of the Custom Vision Service. You can obtain a key by creating a new project at https://customvision.ai and then clicking on the "setting" gear in the top right.

You need to have a training images. In the Resources\Starter\CustomVision.Sample\Images folder are three folders:

* fork
* scissors
* test

The fork and scissors folders contain images of these types of kitchen utensils from different perspectives. The test folder contains an image that will be used to perform the test prediction.

## Create an Object Detection application

Start Visual Studio 2017, Community Edition, open the Visual Studio solution named CustomVision.Sample.sln from the following location:

Resources/Starter/CustomVision.Sample/CustomVision.Sample.sln

Should a "trust" message appear click "Yes".

Once opened, rebuild the solution. Then, within Solution Explorer, double click on Program.cs to open the file.

At the very top of the Program.cs file, use the correct syntax to add references to the two nuget packages:

Microsoft.Azure.CognitiveServices.Vision.CustomVision.Training

Microsoft.Azure.CognitiveServices.Vision.CustomVision.Prediction

To set the application to make use of the objection domain, under the line "// Find the object detection domain" write code that will create a variable named "domains" using the GetDomains method of the trainingApi.

Next, Create a project named Object Detection Project. Under the line "// Create a new project", create a project named Object Detection project and display in the application that a project is being created.

In the second line below, what method will replace the \_ to create the project named Object Detection Project?

Console.WriteLine("Creating new project:");

var project = trainingApi.\_("Object Detection Project", null, objDetectionDomain.Id);

Replace the \_ after "trainingApi." with a method that will display a message in the application that a project is being created.

Create a tag for the scissor images named scissorsTag. Under the lines:

// Make two tags in the new project

var forkTag = trainingApi.CreateTag(project.Id, "fork");

Write code that create a variable named scissorsTag that creates a tag named scissors against the current project.

Upload the scissor images and map them to the scissorsTag. Under the line "// Add all images for scissors" add code that will upload the scissors image and assign the images to the tag of scissorsTag. Hint. Use the code under "// Add all images for fork" as a template for creating the code to upload the scissor images and map them to the scissorsTag.

Train a project named Object Detection Project. Under the line "// Now there are images with tags start training the project", the next step is to train the project and have the application display that the project is being trained.

In the second line below, what method will replace the \_ to train the project?

Console.WriteLine("\tTraining");

var iteration = trainingApi.\_(project.Id);

Replace the \_ with a method that will display a message in the application that a project is being trained.

Build and run the solution. The training and prediction of the images can take 2 minutes. The prediction results appear on the console.

If you need help Start Visual Studio 2017, Community Edition, open the Visual Studio solution named CustomVision.Sample.sln in the solution sub-directory of where this lab is located Resources/Solution/CustomVision.Sample/CustomVision.Sample.sln

# Lab 3 Creating custom language models using LUIS

Now that we know what LUIS is, we'll want to plan our LUIS app. Next, we'll be creating a bot ("PictureBot") that returns images based on our search, that we can then share or order. We will need to create intents that trigger the different actions that our bot can do, and create entities that require different actions. For example, an intent for our PictureBot may be "SearchPics" and it triggers Azure Search service to look for photos, which requires a "facet" entity to know what to search for.

hese are the steps you will generally take when creating LUIS applications:

* Add intents
* Add utterances
* Add entities
* Improve performance using phrase lists and patterns
* Train and test
* Review endpoint utterances
* Publish

## Create the LUIS service

In the Portal, hit Create a resource and then enter LUIS in the search box and choose Language Understanding.

This will lead you to fill out a few details for the API endpoint you'll be creating, choosing the API you're interested in and where you'd like your endpoint to reside, as well as what pricing plan you'd like. Put it in a location that is close to you and available. The free tier is sufficient for this lab.

Once you have created your new API subscription, you can grab the key from the appropriate section of the blade and add it to your list of keys.

## Add intelligence to your applications with LUIS

Let's look at how we can use LUIS to add some natural language capabilities. LUIS allows you to map natural language utterances (words/phrases/sentences the user might say when talking to the bot) to intents (tasks or actions the user wants to perform). For our application, we might have several intents: finding pictures, sharing pictures, and ordering prints of pictures, for example. We can give a few example utterances as ways to ask for each of these things, and LUIS will map additional new utterances to each intent based on what it has learned.

Navigate to https://www.luis.ai (if you created a key in a Europe region, you will need to create your application at https://eu.luis.ai/). From the main page, click the "Create LUIS app" button, select your country on the next page, accept the license agreement, and click "Create new app" button on the next page. Give it a name (I chose "PictureBotLuisModel") and set the Culture to "English". You can optionally provide a description. Then click "Done".

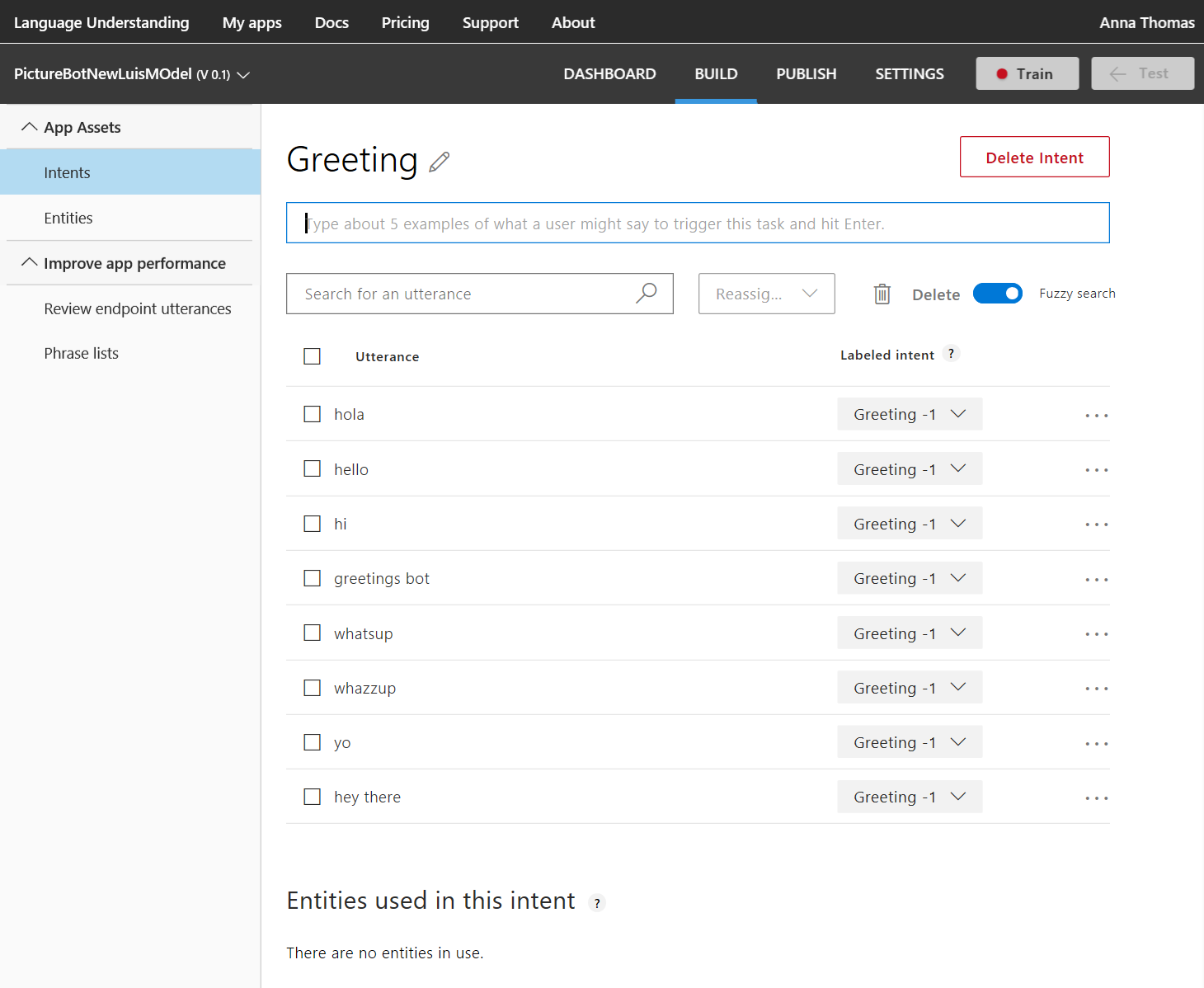
You will be taken to the Build section for your new app. Note that there is one intent called "None". Random utterances that don't map to any of your intents may be mapped to "None".

We want our bot to be able to do the following things:

* Search/find pictures
* Share pictures on social media
* Order prints of pictures
* Greet the user (although this can also be done other ways, as we will see later)

Let's create intents for the user requesting each of these. Click the "Create new intent" button.

Name the first intent "Greeting" and click "Done". Then give several examples of things the user might say when greeting the bot, pressing "Enter" after each one.



Let's see how to create an entity. When the user requests to search the pictures, they may specify what they are looking for. Let's capture that in an entity.

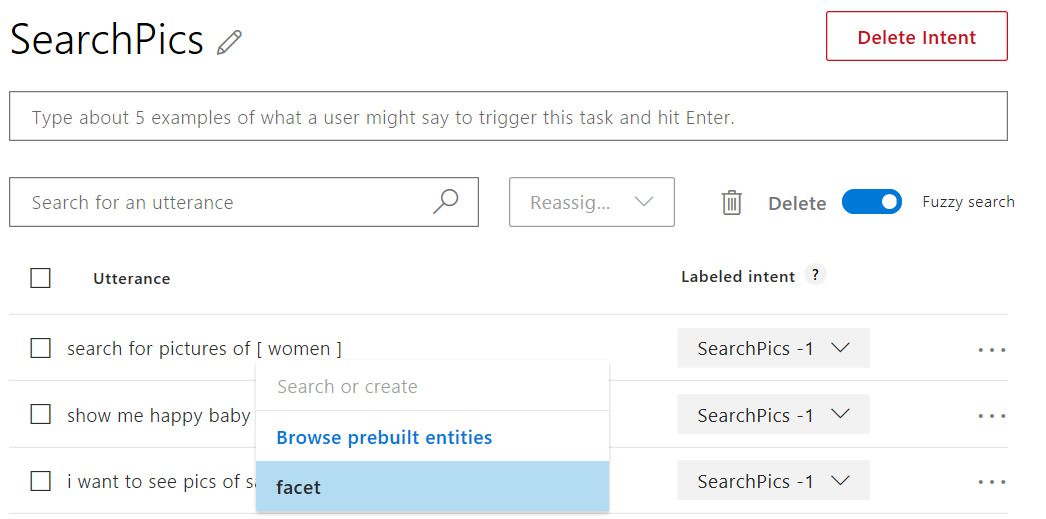
Click on "Entities" in the left-hand column and then click "Create new entity". Give it an entity name "facet" and entity type "Simple". Then click "Done".

Next, click "Intents" in the left-hand sidebar and then click the "Create new intent" button. Give it an intent name of "SearchPics" and then click "Done".

Just as we did for Greetings, let's add some sample utterances (words/phrases/sentences the user might say when talking to the bot). People might search for pictures in many ways. Feel free to use some of the utterances below, and add your own wording for how you would ask a bot to search for pictures.

* Find outdoor pics
* Are there pictures of a train?
* Find pictures of food.
* Search for photos of boys playing
* Please give me pics of business women
* Show me beach pics
* I want to find dog photos
* Search for pictures of men indoors
* Show me pictures of men wearing glasses
* I want to see pics of sad boys
* Show me happy baby pics

Once we have some utterances, we have to teach LUIS how to pick out the search topic as the "facet" entity. Whatever the "facet" entity picks up is what will be searched. Hover and click the word (or click consecutive words to select a group of words) and then select the "facet" entity.



Next, click "Intents" in the left sidebar and add two more intents:

* Name one intent "SharePic". This might be identified by utterances like "Share this pic", "Can you tweet that?", or "post to Twitter".
* Create another intent named "OrderPic". This could be communicated with utterances like "Print this picture", "I would like to order prints", "Can I get an 8x10 of that one?", and "Order wallets".

When choosing utterances, it can be helpful to use a combination of questions, commands, and "I would like to..." formats.

Finally, you need to add some sample utterances to the "None" intent. This helps LUIS label when things are outside the scope of your application. Add things like "I'm hungry for pizza", "Search videos", etc. You should have about 10-15% of your app's utterances within the None intent.

We are now ready to train our model. Click "Train" in the top right bar. This builds a model to do utterance --> intent mapping with the training data you've provided. Training is not always immediate. Sometimes, it gets queued and can take several minutes.

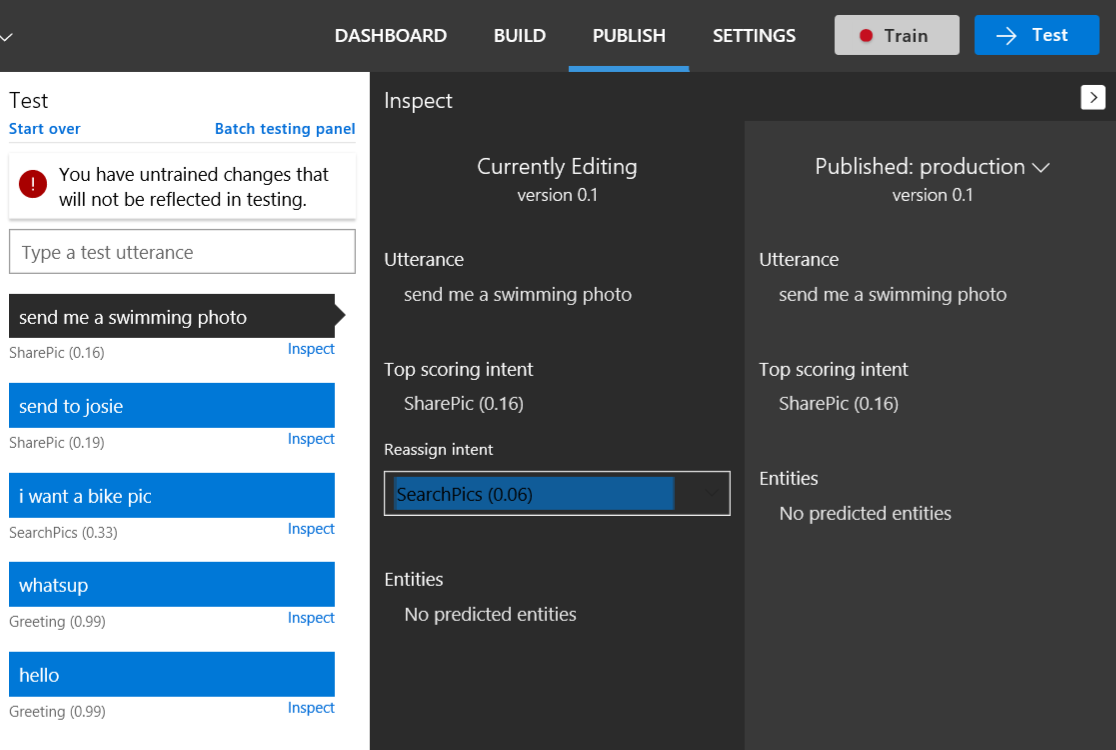
Next, click on "Manage" in the top bar. You'll have several options on the left side of the window (Application Information, Keys and Endpoints, Publish Settings, Versions, Collaborators). You can read more about the various publish options here.

Finally, select or add (by clicking the "Add Key" button towards the bottom of the page on the "Keys and Endpoints" tab) the endpoint key that you set up earlier, or follow the link to add a key from your Azure account (for testing and learning purposes, you can also just use the "Starter\_Key", which is why creating the LUIS service was optional). You can leave the endpoint slot as "Production". Then click "Publish" in the top bar. You'll have the option to publish to your "Production" or "Staging" endpoint. Select "Production", and read about the reasons for the two endpoints. Finally, click "Publish".

Publishing creates an endpoint to call the LUIS model. The URL will be displayed, which will be explained in a later lab. For now, you can copy the Endpoint URL and add it to your list of keys.

Click on "Test" in the top right bar. Try typing a few utterances and see the intents returned. Familiarize yourself with interactive testing and reviewing endpoint utterances as you may want to do this now or in a future lab.

One quick example is shown below. I have noticed that my model incorrectly assigned "send me a swimming photo" as SharePic, when it should be SearchPics. I reassigned the intent.



Now I need to retrain my app by selecting the Train button. I then tested the same utterance and compared the results between my recently trained and previously published model. Remember, you'll have to republish your model to see updates in the application that uses the model.

You can also test your published endpoint in a browser. Copy the Endpoint URL. To open this URL in your browser, set the URL parameter &q= to your test query. For example, append Find pictures of dogs to your URL, and then press Enter. The browser displays the JSON response of your HTTP endpoint.

# Lab 4 Creating search index using Azure Search

Typical Workflow:

* Provision service. You can create or provision an Azure Search service from the portal or with PowerShell.
* Create an index. An index is a container for data, think "table". It has schema, CORS options, search options. You can create it in the portal or during app initialization.
* Index your data. There are two ways to populate an index with your data. The first option is to manually push your data into the index using the Azure Search REST API or .NET SDK. The second option is to point a supported data source to your index and let Azure Search automatically pull in the data on a schedule.
* Search an index. When submitting search requests to Azure Search, you can use simple search options, you can filter, sort, project, and page over results. You have the ability to address spelling mistakes, phonetics, and Regex, and there are options for working with search and suggest. These query parameters allow you to achieve deeper control of the full-text search experience.

## Create an Azure Search Service

Within the Azure Portal, click Create a resource, enter "azure search" in the search bar, and click Azure Search->Create.

Once you click this, you'll have to fill out a few fields as you see fit. For this lab, the "F" free tier is sufficient. You are only able to have one Free Azure Search instance per subscription, so if you or another member on your subscription have already done this, you will need to use the "Basic" pricing tier. Use the one Resource Group for all of the labs in this workshop. If you already have a resource group for this workshop, just use that one. Put in one of the following locations: West US, West US 2, East US, West Europe, Southeast Asia.

Once creation is complete, open the panel for your new search service.

## Create an Azure Search Index

An index is a persistent store of documents and other constructs used by an Azure Search service. An index is like a database that holds your data and can accept search queries. You define the index schema to map to the structure of the documents you wish to search, similar to fields in a database. These fields can have properties that tell things such as if it is full text searchable, or if it is filterable. You can populate content into Azure Search by programmatically pushing content or by using the Azure Search Indexer (which can crawl common datastores for data).

For this lab, we will use the Azure Search Indexer for Cosmos DB to crawl the data in the Cosmos DB collection.

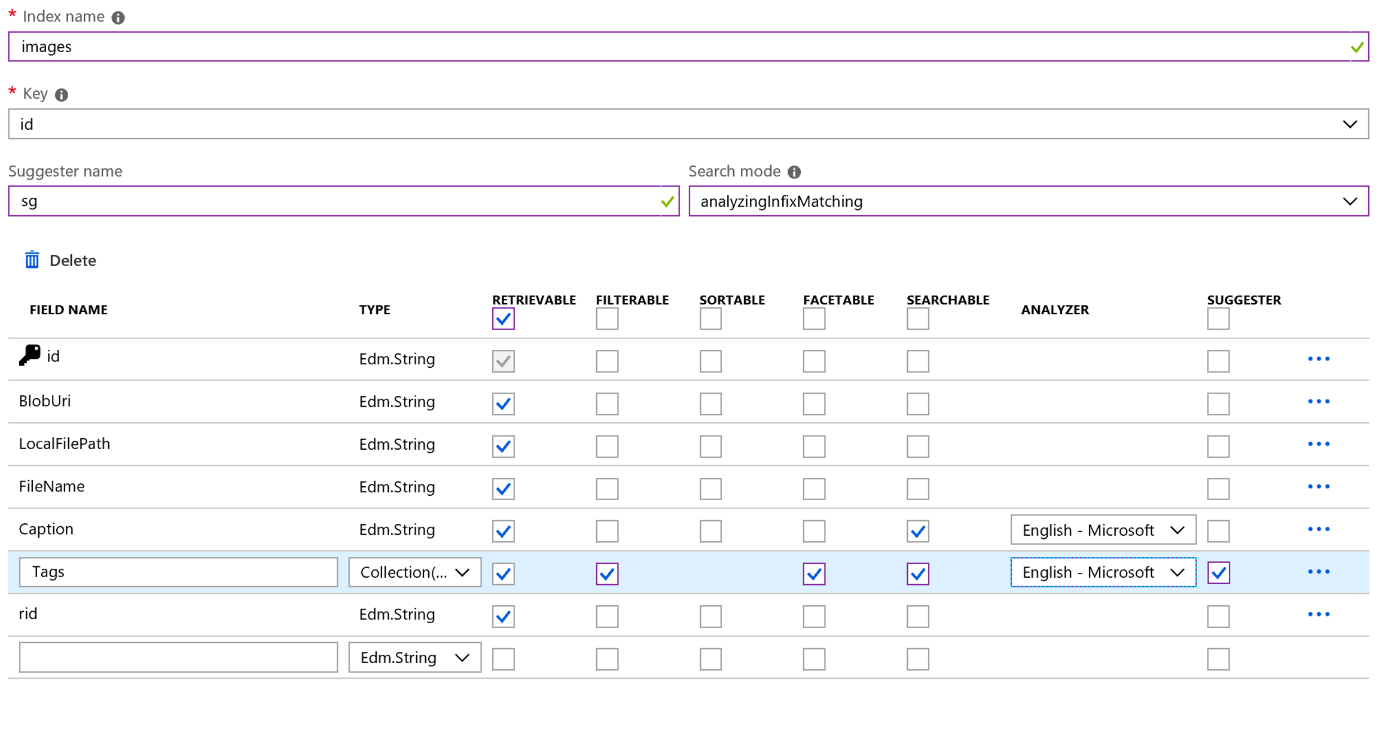
Within the Azure Search blade you just created, click Import Data->Data Source->Cosmos DB. Once you click this, choose a name for the Cosmos DB data source. If you completed the previous lab, lab1-computer\_vision, choose the Cosmos DB account where your data resides as well as the corresponding Container and Collections. If you did not complete the previous lab find a friendly neighbor that is willing to provide their connection string. For both, the Database should be "images" and the Collection should be "metadata".

Click OK or Next twice to skip Cognitive Search. At this point Azure Search will connect to your Cosmos DB container and analyze a few documents to identify a default schema for your Azure Search Index. After this is complete, you can set the properties for the fields as needed by your application (e.g. you can "Customize target index").

Azure Portal Interface may be different, but the information below will guide you in this process.

* Update the Index name to: images
* Update the Key to: id (which uniquely identifies each document)
* Create a Suggester to set the fields that will be used for type ahead, allowing the user to type parts of a word where Azure Search will look for best matches in these fields. To learn more about suggestors and how to extend your searches to support fuzzy matching, which allows you to get results based on close matches even if the user misspells a word, check out this example. Enter a Suggester Name sg and choose the only one option in the Search mode. Set Tags to be the fields to look for term suggestions
* Set all fields to be Retrievable (to allow the client to retrieve these fields when searched)
* Set the fields Tags to be Filterable (to allow the client to filter results based on these values)
* Set the fields Tags to be Facetable (to allow the client to group the results by count, for example for your search result, there were "5 pictures that had a Tag of "beach")
* Set the fields Caption and Tags to be Searchable (to allow the client to do full text search over the text in these fields). Set the fields Caption and Tags to use the English-Microsoft Analyzer option.

At this point we will configure the Azure Search Analyzers. At a high level, you can think of an analyzer as the thing that takes the terms a user enters and works to find the best matching terms in the Index. Azure Search includes analyzers that are used in technologies like Bing and Office that have deep understanding of 50+ languages.



Click OK or Next to complete the configuration of the Indexer.

Next, we configure how to "Import your data". You could set at schedule for how often the Indexer should check for changes, however, for this lab we will just run it once.

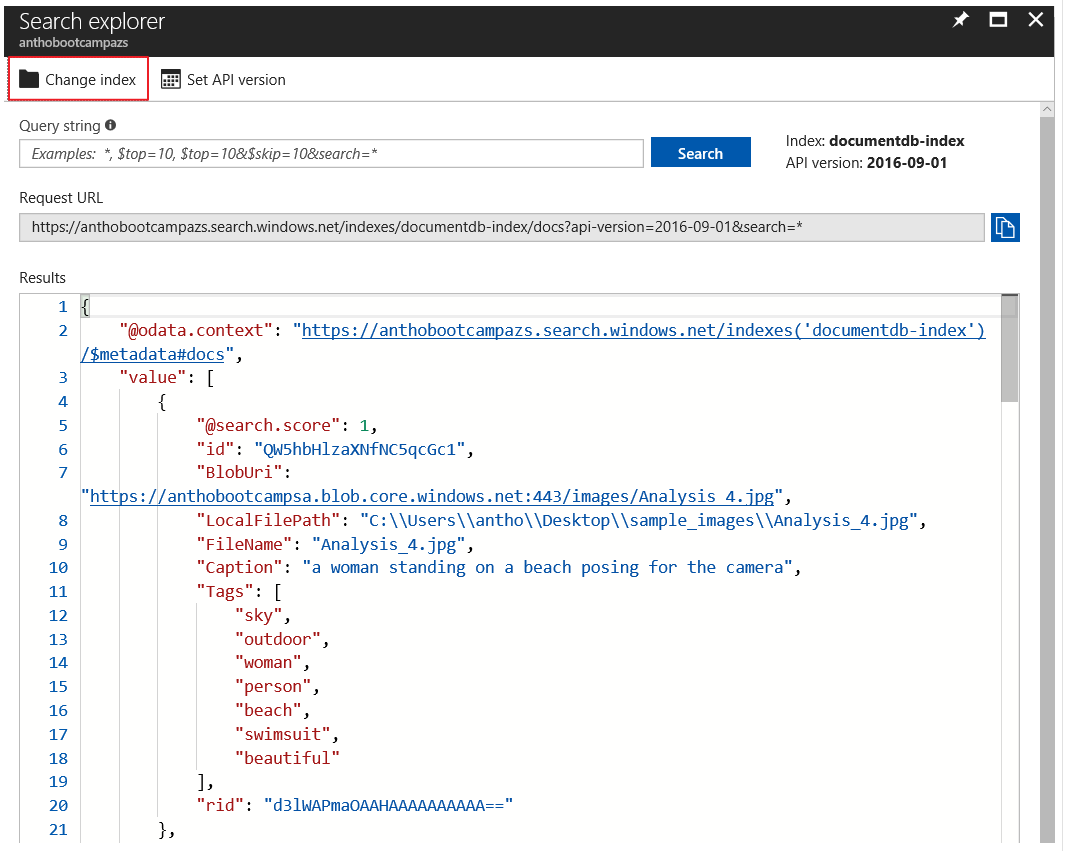
* Click Advanced Options and choose to Base 64 Encode Keys to ensure that the ID field only uses characters supported in the Azure Search key field.
* Click OK or Submit to start the Indexer job that will start the importing of the data from the Cosmos DB database.

## Query the Search Index

At this point we can try searching the index.

Click Search Explorer and in the resulting blade choose your Index if it is not already selected.

Click Search to search for all documents. Try searching for "water", or something else, and use ctrl+click to select and view the URLs. Were your results what you expected?



In the resulting json, you'll see a number after @search.score. Scoring refers to the computation of a search score for every item returned in search results. The score is an indicator of an item's relevance in the context of the current search operation. The higher the score, the more relevant the item. In search results, items are rank ordered from high to low, based on the search scores calculated for each item.

Azure Search uses default scoring to compute an initial score, but you can customize the calculation through a scoring profile. There is an extra lab at the end of this workshop if you want to get some hands on experience with using term boosting for scoring.

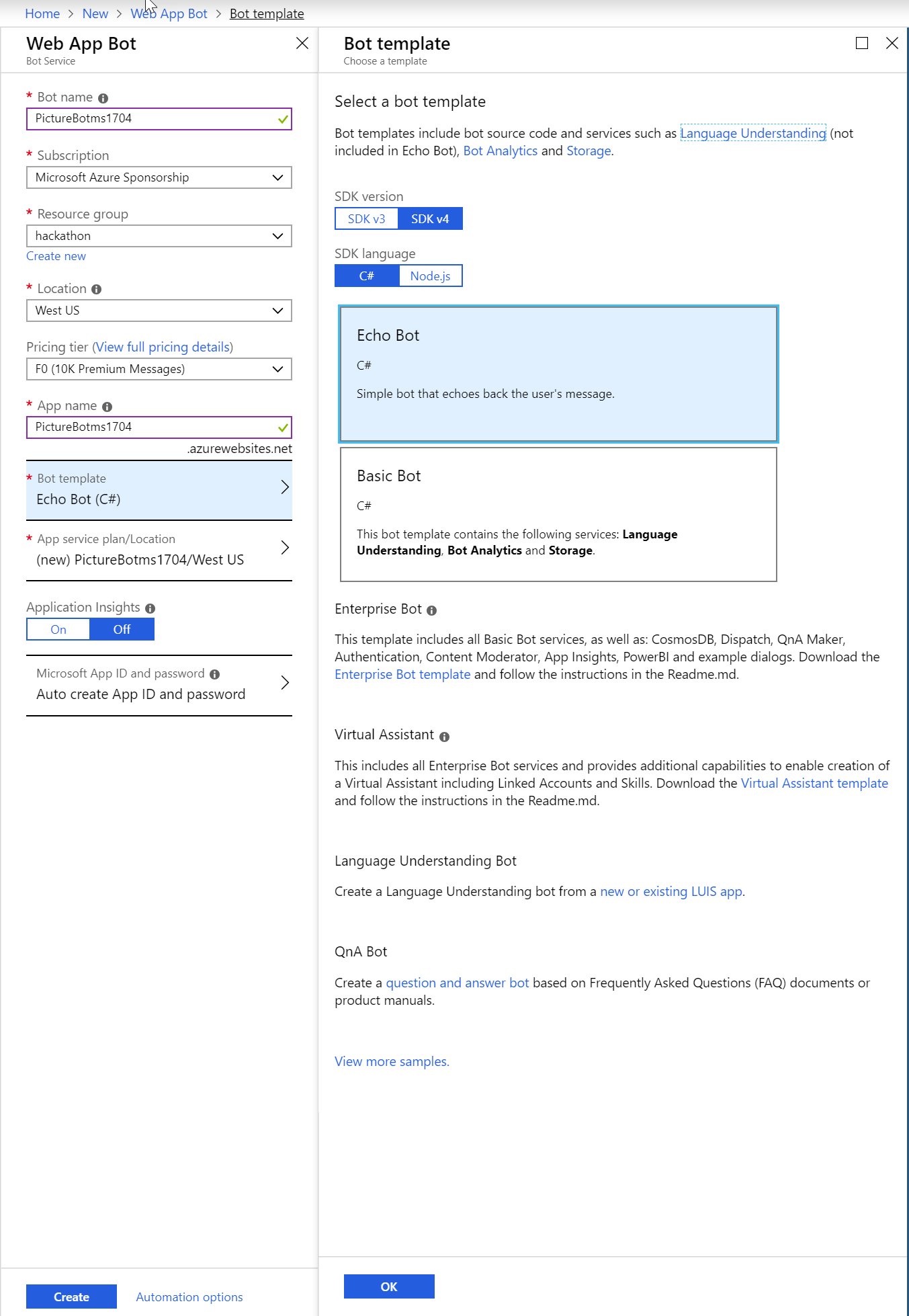
# Lab 5 Developing Intelligent Bots

We will be developing a bot using the latest .NET SDK (v4). To get started, we'll need to download the Bot Framework Emulator, and we'll need to create a Web App Bot and get the source code.

## Create an Azure Web App Bot

A bot created using the Microsoft Bot Framework can be hosted at any publicly-accessible URL. For the purposes of this lab, we will register our bot using Azure Bot Service.

Navigate to the portal. In the portal, click "Create a resource" and search for "bot". Select "Web App Bot", and click create. For the name, you'll have to create a unique identifier. I recommend using something along the lines of PictureBot[i][n] where [i] is your initials and [n] is a number (e.g. mine would be PictureBotamt6). Put in the region that is closest to you. For pricing tier, select F0, as that is all we will need for this workshop. Select a Bot template of SDK v4 for Echo Bot in C#, we will update this code to be our PictureBot. Configure a new App service plan (put it in the same location as your bot). You can choose to turn Application Insights on or off. Do not change or click on "Auto create App ID and password", we will get to that later. Click create.



When it's deployed, go to the Web App Resource. You have just deployed a very simple EchoBot with the Echo Bot template. If you wish, you can select the "Test in WebChat" tab once it's deployed, and see what it is capable of doing.

Since this isn't the bot we set out to build, the next thing you need to do is download the source code. We'll edit this throughout the next few labs, and then we'll publish it back to this service.

Navigate to the "Build" tab within the portal, and select "Download Bot source code". Save it somewhere convenient and extract all the files from the zip file.

While you've got the Azure portal open, now is also a good time to find and save the botFilePath and botFileSecret located in App Service Settings > Application Settings > Application settings section of your Web App Bot service.

## Creating a simple bot and running it

Navigate to the solution file for the Web App Bot you just created, and open it in Visual Studio (in future bots, you can use Visual Studio Code, but for the purposes of these labs, you should use Visual Studio). Spend some time looking at all of the different things you get built in from the Echo Bot template. We won't spend time explaining every single file, but we highly recommend spending some time later working through and reviewing this sample (and the other Web App Bot sample - Basic Bot), if you have not already. It contains important and useful shells needed for bot development. For the purposes of this lab, we will modify aspects of the template to fit our needs, which is what you would likely do in the real world as well.

Start by right-clicking on the Solution and selecting "Build". This will load up the packages. Next, in the "appsettings.json file, update it to include the following, adding your bot service information as well:

{

"botFilePath": "YourBotFilePath",

"botFileSecret": "YourBotFileSecret"

}

Next, we'll focus on the NuGet packages we need. Right-click on the solution in Solution Explorer and select "Manage NuGet Packages for Solution."

In the "Installed" tab, update the following packages in order to 4.1.5 (this may already be done):

* Microsoft.Bot.Configuration
* Microsoft.Bot.Schema
* Microsoft.Bot.Connector
* Microsoft.Bot.Builder
* Microsoft.Bot.Builder.Integration.AspNet.Core

Next, click the "Browse" tab, and install all of the packages listed below. Make sure you check the box "Include prerelease" and are on the "Browse" tab:

* Microsoft.Bot.Builder.Azure
* Microsoft.Bot.Builder.AI.Luis
* Microsoft.Bot.Builder.Dialogs
* Microsoft.Azure.Search

Finally, in the solution explorer, navigate to Dependencies > NuGet and remove the following packages:

* AsyncUsageAnalyzers
* StyleCop.Analyzers

So now that we've updated our base shell to support the naming and NuGet packages we'll use throughout the rest of the labs, we're ready to start adding some custom code. First, we'll just create a simple "Hello world" bot that helps you get warmed up to building bots with the V4 SDK.

An important concept is the "turn", used to describe a message to a user and a response from the bot. For example, if I say "Hello bot" and the bot responds "Hi, how are you?" that is one turn. Check in the image below how a turn goes trought the multiple layers of a bot application.

Bots Concepts

