AI-100-DesignImplementAzureAISol

Lab files for AI100T01A ILT Course

Lab 1: Meeting the Technical Requirements

In this lab, we will introduce our workshop case study and setup tools on your local workstation and in your Azure instance to enable you to build tools within the Microsoft Cognitive Services suite.

Lab 2: Implement Computer Vision Capabilities for a Bot

This hands-on lab guides you through creating an intelligent console application from end-to-end using Cognitive Services (specifically the Computer Vision API). We use the ImageProcessing portable class library (PCL), discussing its contents and how to use it in your own applications.

Lab 3: Basic Filtering Bot

In this lab, we will be setting up an intelligent bot from end-to-end that can respond to a user's chat window text prompt. We will be building on what we have already learned about building bots within Azure, but adding in a layer of custom logic to give our bot more bespoke functionality.

This bot will be built in the Microsoft Bot Framework. We will evoke the architecture that enables the bot interface to receive and respond with textual messages, and we will build logic that enables our bot to respond to inquiries containing specific text.

We will also be testing our bot in the Bot Emulator, and addressing the middleware that enables us to perform specialized tasks on the message that the bot receives from the user.

We will evoke some concepts pertaining to Azure Cognitive Search, and Microsoft's Language Understanding Intelligent Service (LUIS), but will not implement them in this lab.

Lab 4: Log Bot Chat

In the previous lab, we started with an echo bot project and modified the code to suit our needs. Now, we wish to log chats with our bots to enable our customer service team to follow up to inquiries, determine if the bot is performing in the expected manner, and to analyze customer data.

This hands-on lab guides you through enabling various logging scenarios for your bot solutions.

In the advanced analytics space, there are plenty of uses for storing log conversations. Having a corpus of chat conversations can allow developers to:

1. Build question and answer engines specific to a domain.
2. Determine if a bot is responding in the expected manner.
3. Perform analysis on specific topics or products to identify trends.

In the course of the following labs, we'll walk through how we can enable chat logging and intercept messages. We will evoke some of the various ways we might also store the data, although data solutions are not within the scope of this workshop.

Lab 5: QnA Maker

In this lab you will use the Microsoft QnA Maker application to create a knowledgebase, publish it and then consume it in your bot.

Lab 6: Implement the LUIS model

We're going to build an end-to-end scenario that allows you to pull in your own pictures, use Cognitive Services to find objects and people in the images, and obtain a description and tags. We'll later build a Bot Framework bot using LUIS to allow easy, targeted querying.

Lab 7: Integrate LUIS into a bot with Dialogues

Now that our bot is capable of taking in a user's input and responding based on the user's input, we will give our bot the ability to understand natural language with the LUIS model we built in Lab 6

Lab 8: Detect User Language

In this lab we will add the ability for your bot to detect languages from user input.

If you have trained your bot or integrated it with QnA Maker but have only done so using only one particular language, then it makes sense to inform users of that fact.

Lab 9: Test Bots in DirectLine

This hands-on lab guides you through some of the basics of testing bots. This workshop demonstrates how you can perform functional testing (using Direct Line).

Lab 1: Technical Requirements

Introduction

In this lab, we will introduce our workshop case study and setup tools on your local workstation and in your Azure instance to enable you to build tools within the Microsoft Cognitive Services suite.

Workshop Case Study

You've been assigned a new customer, Adventure Works LLC, which sells bicycles and bicycle equipment to its customers.

Adventure Works Cycles is a large multinational manufacturing company. It manufactures and sells bicycles and bicycle components to the North American, European and Asian commercial markets through both an internet channel and a reseller distribution network. Its base operations are in Kirkland, Washington with 290 employees, and there are several regional sales teams located throughout their market base.

Coming off a successful fiscal year, Adventure Works is looking to increase its revenue by targeting additional sales to their existing customers. In the last year, the marketing department started an initiative where they manually collected product ratings data for the Adventure Works products at various trade shows and racing events. This data is currently held in a Microsoft Excel file.

They have been able to prove that with the information collected, they are able to sell additional products to the existing client base through a manual analysis of the rating���s results. The Marketing team wanted to scale this by creating an online version of the survey on the Adventure Works website, but the Sales department heard of their effort and created an online survey. The take up of the survey on this platform has been very poor and the fields do not align with the data collected in the Excel files.

The Marketing department strongly believe that they can use the ratings data to make recommendations about other products as an upsell opportunity for the business. However, placing the survey in the website is proving to be a poor channel to harvest the data required and they are seeking advice on how this situation can be improved. In addition, the team are aware that performing the analysis manually will prove too difficult as more customers.

Adventure Works aims to seamlessly scale up to handle large inquiry volumes of customers speaking various languages. Additionally, they wish to create a scalable customer service platform to gain more insight about customers' needs, problems, and product ratings.

In addition, the Customer Services department would like to offload some of their customer support function to an interactive platform. The intention is to reduce the workload on the staff and increase customer satisfaction by answering common questions quickly.

Solution

The interactive platform is envisioned as a bot that will consist of the following functionality:

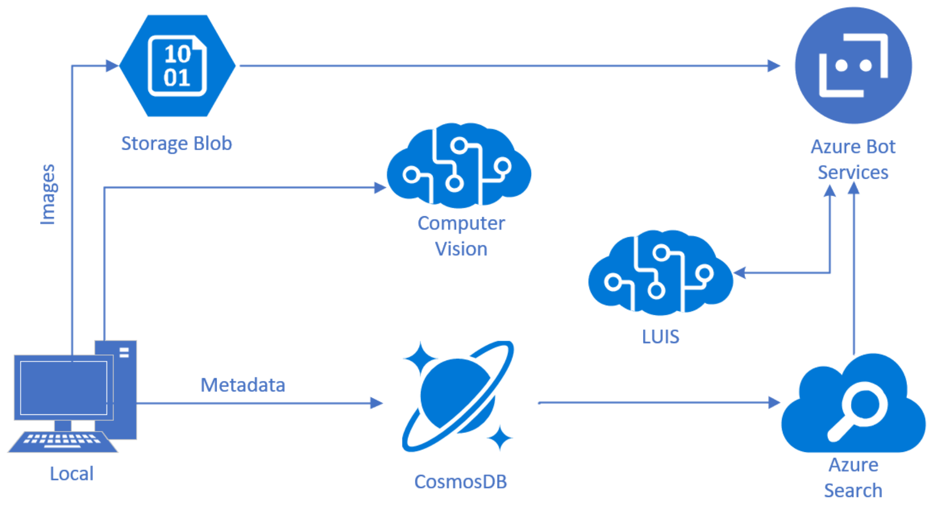
- Detect customer language (responding that only English is supported at this time)
- Monitor sentiment of the user
- Allow image uploading and determine if object is a bicycle
- Integrate FAQ into a chatbot
- Determine the user���s intentions based on entered text in the bot chat
- Log the chat bot session for later review

We will build a simple C# application that allows you to ingest pictures from your local drive, then invoke the Computer Vision API to analyze the images and obtain tags and a description.

In the continuation of this lab throughout the lab, we'll show you how to build a Bot Framework bot to interact with customers' text inquiries. We will then demonstrate a quick solution for integrating existing Knowledge Bases and FAQs into a bot framework with the QnA Maker. Finally, we'll extend this bot with LUIS to automatically derive intent from your queries and use those to respond to your customers' text requests intelligently.

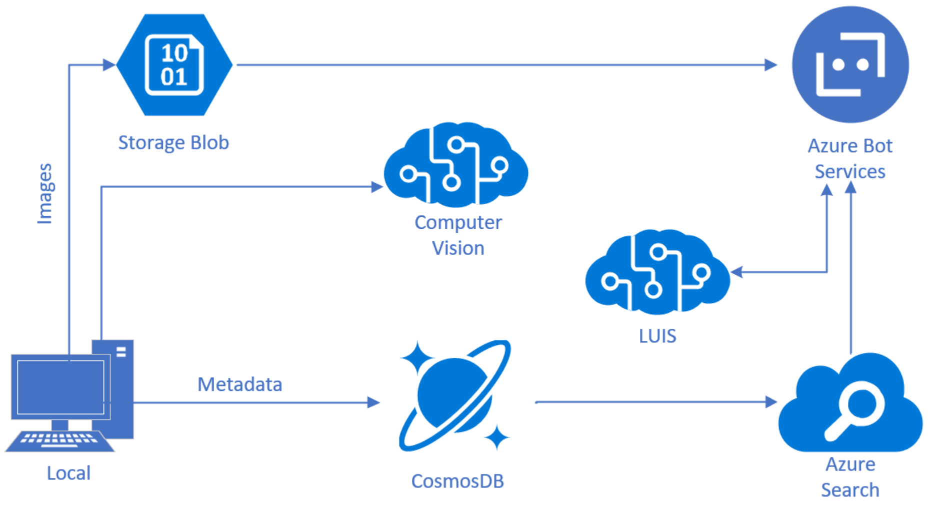
We will only provide context for using Bing Search to enable customers to access other data during interactions with the bot, but will not implement these scenarios during the lab. Participants are invited to read further about the Bing Web Search services.

While out of scope for this lab, this architecture integrates Azure's data solutions manage storage of images and metadata in this architecture, through Blob Storage and Cosmos DB.



Architecture

Your team recently presented a potential architecture (below) that Adventure Works approved:



- Computer Vision allows uploading images, detects contents
- QnA Maker facilitating bot interactions from a static knowledge base
- Text Analytics enables language detection
- LUIS (Language Understanding Intelligent Service) extracts intent and entities from text
- Azure Bot Service connector service to enable chatbot interface to leverage app intelligence

Next Steps

- Lab 01-02: Technical Requirements

Lab 1: Meeting the Technical Requirements

Setting technology, environments and keys

This lab is meant for an Artificial Intelligence (AI) Engineer or an AI Developer on Azure. To ensure you have time to work through the exercises, there are certain requirements to meet before starting the labs for this course.

You should ideally have some previous exposure to Visual Studio. We will be using it for everything we are building in the labs, so you should be familiar with how to use it to create applications. Additionally, this is not a class where we teach code or development. We assume you have some familiarity with C# (intermediate level - you can learn here and here).

Account Setup

> **Note** You can use different environments for completing this lab. Your instructor will guide you through the necessary steps to get the environment up and running. This could be as simple as using the computer you are logged into in the classroom or as complex as setting up a virtualized environment. The labs were created and tested using the Azure Data Science Virtual Machine (DSVM) on Azure and as such, will require an Azure account to use.

Setup 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.

Environment Setup

These labs are intended to be used with the .NET Framework using Visual Studio 2019 runnning on a Microsoft Windows operating system. While there is a version of Visual Studio for Mac OS, certain features in the sample code are not supported on the Mac OS platform. As a result, there is a hosted lab option available using a virtual machine solution. Your instructor will have the details on using the VM solution. The original workshop was designed to be used, and was tested with, the Azure Data Science Virtual Machine (DSVM). Only premium Azure subscriptions can actually create a DSVM resource on Azure but the labs can be completed with a local computer running Visual Studio 2019 and the required software downloads listed throughout the lab steps.

Urls and Keys Needed

Over the course of this lab, we will collect a variety of 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. Not all of these will be populated in this lab.

Keys

- Cognitive Services API Url:
- Cognitive Services API key:
- LUIS API Endpoint:
- LUIS API Key:
- LUIS API App ID:
- Bot App Name:
- Bot App ID:
- Bot App Password:
- Azure Storage Connection String:
- Cosmos DB Url:
- Cosmos DB Key:
- DirectLine Key:

Azure Setup

In the following steps, you will configure the Azure environment for the labs that follow.

Cognitive Services

While the first lab focuses on the Computer Vision Cognitive Service, Microsoft Azure allows you to create a cognitive service account that spans all services, or you can elect to create a cognitive service account for an individual service. In the following steps, you will create a single Azure resource that contains all available cognitive services endpoints.

1. Open the Azure Portal

2. Select **+ Create a Resource** and then enter **cognitive services** in the search box

3. Choose **Cognitive Services** from the available options, then select **Create**

    > **Note** Again to reiterate, you can create specific cognitive services resources or you can create a single resource that contains all the endpoints.

4. Type a name of your own choosing

5. Select your subscription and resource group

6. For the pricing tier, select **S0**

7. Check the confirmation checkbox

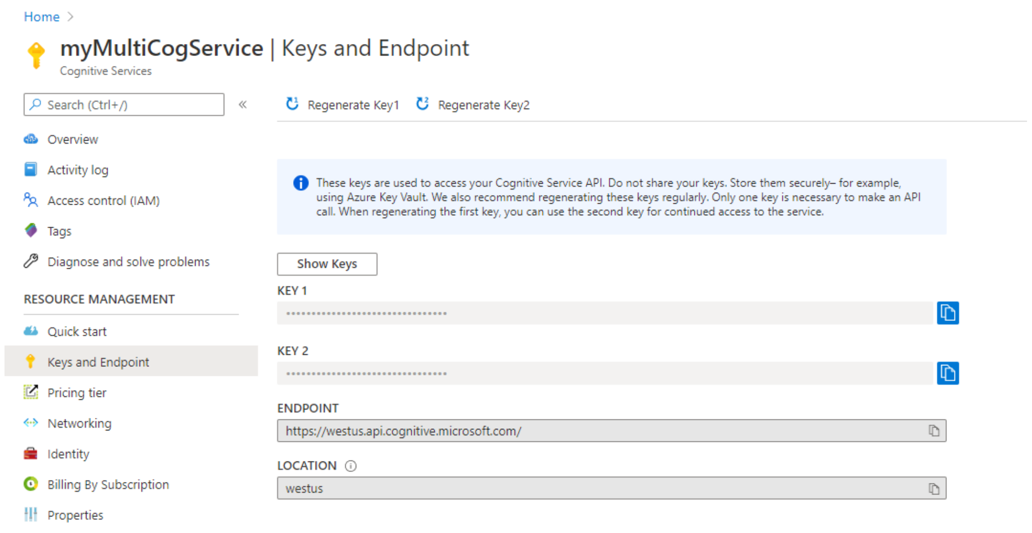
    > [!Note] Microsoft updates the Azure portal and services on a regular basis. These steps contained the appropriate items at the time of writing but options and dialogs may differ if changes are made to Azure. Check with your instructor for any anomalies that you may encounter.

8. Select **Review + create**

9. Once validation passes, select **Create**

10. Navigate to the new resource, under the **Resource Management** section in the left toolbar, select **Keys and Endpoints**

11. Copy the **API Key** and the **url of the endpoint** to your notepad



Home >

myMultiCogService | Keys and Endpoint

Cognitive Services

These keys are used to access your Cognitive Service API. Do not share your keys. Store them securely– for example, using Azure Key Vault. We also recommend regenerating these keys regularly. Only one key is necessary to make an API call. When regenerating the first key, you can use the second key for continued access to the service.

KEY 1: ••••••••••••••••••••••••••••••••

KEY 2: ••••••••••••••••••••••••••••••••

ENDPOINT: https://westus.api.cognitive.microsoft.com/

LOCATION: westus

Azure Storage Account

1. In the Azure Portal, select **+ Create a Resource** and then enter **storage** in the search box

2. Choose **Storage account** from the available options, then select **Create**

3. Select your subscription and resource group

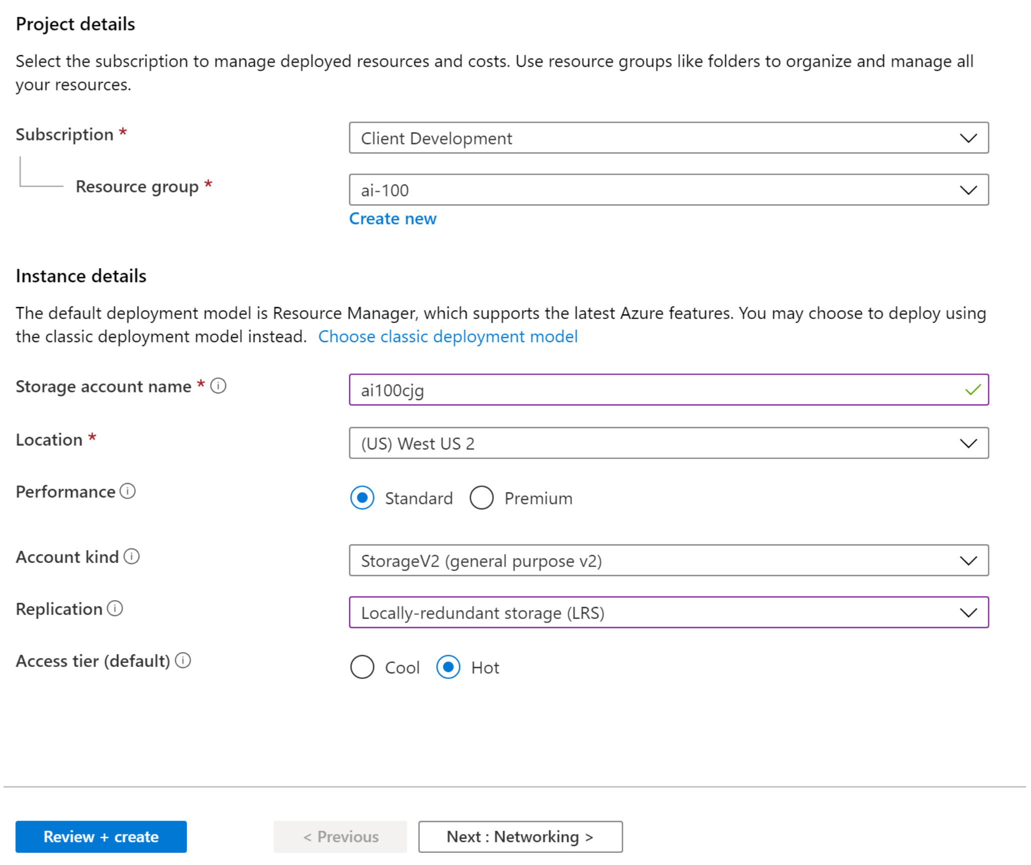
4. Type a name for your account, using your initials to make it unbique. **ai100storagego**

5. For the location, select the same as your resource group

6. Performance should be **Standard**

7. Account kind should be **StorageV2 (general purpose v2)**

8. For replication, select **Locally-redundant storage (LRS)**



Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *	Client Development
Resource group *	ai-100

Instance details

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. Choose classic deployment model

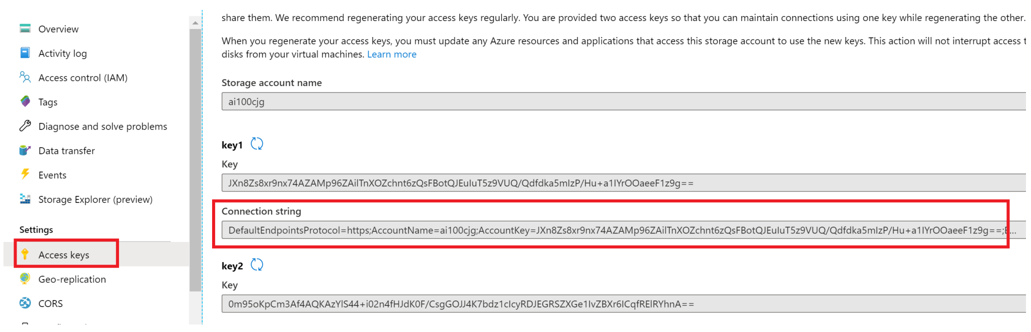
Storage account name *	ai100cjg ✓
Location *	(US) West US 2
Performance	[✓ Standard] Premium
Account kind	StorageV2 (general purpose v2)
Replication	Locally-redundant storage (LRS)
Access tier (default)	Cool [✓ Hot]

9. Select **Review + create**

10. Select **Create**

11. Navigate to the new resource, select **Access Keys**

12. Copy the **Connection string** to your notepad



share them. We recommend regenerating your access keys regularly. You are provided two access keys so that you can maintain connections using one key while regenerating the other.

When you regenerate your access keys, you must update any Azure resources and applications that access this storage account to use the new keys. This action will not interrupt access t
disks from your virtual machines. Learn more

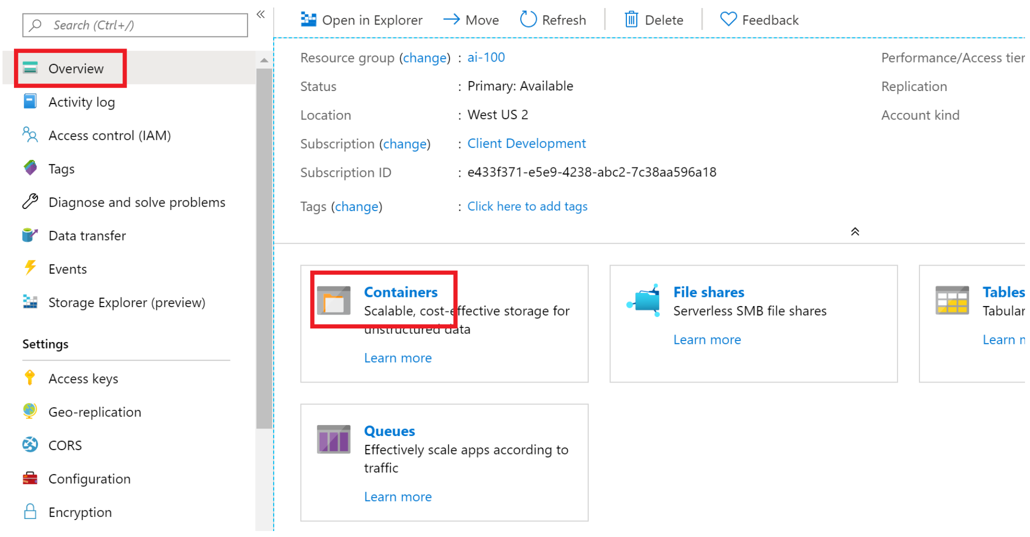
Storage account name
ai100cjg

key1
Key
JXn8Zs8xr9nx74AZAMp96ZAilTnXOZchnt6zQsFBotQJEuluT5z9VUQ/Qdfdka5mlzP/Hu+a1lYrOOaeeF1z9g==

Connection string
DefaultEndpointsProtocol=https;AccountName=ai100cjg;AccountKey=JXn8Zs8xr9nx74AZAMp96ZAilTnXOZchnt6zQsFBotQJEuluT5z9VUQ/Qdfdka5mlzP/Hu+a1lYrOOaeeF1z9g==;E...

key2
Key
0m95oKpCm3Af4AQKAzYlS44+i02n4fHJdK0F/CsgGOJJ4K7bdz1clcyRDJEGRSZXGe1lvZBXr6lCqfREIRYhnA==

13. Select **Overview**, then select **Containers**



Resource group (change) : ai-100
Status : Primary: Available
Location : West US 2
Subscription (change) : Client Development
Subscription ID : e433f371-e5e9-4238-abc2-7c38aa596a18
Tags (change) : Click here to add tags

Performance/Access tier
Replication
Account kind

Containers
Scalable, cost-effective storage for unstructured data
Learn more

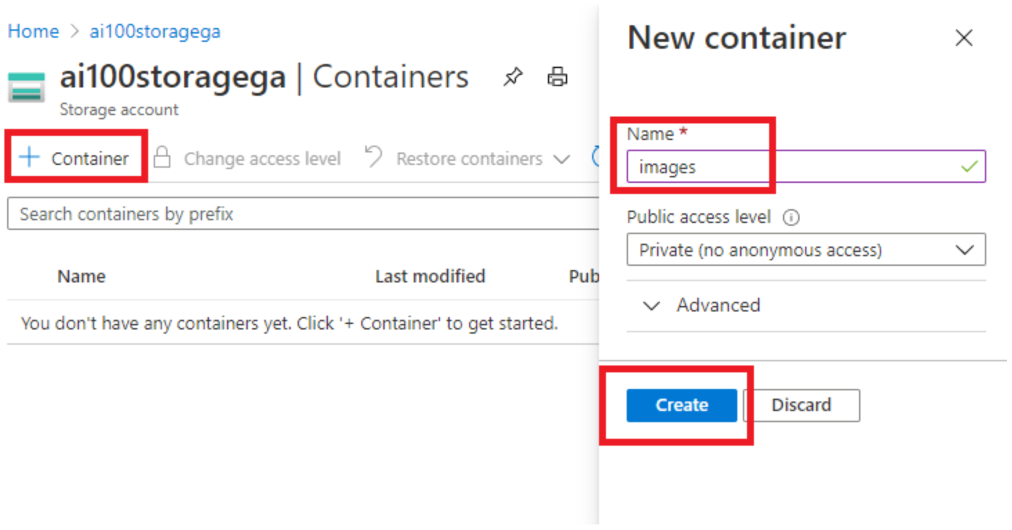
File shares
Serverless SMB file shares
Learn more

Tables
Tabular
Learn m

Queues
Effectively scale apps according to traffic
Learn more

14. Select **+ Container**

15. For the name, type **images**



You don't have any containers yet. Click '+ Container' to get started.

Name *
images
Public access level
Private (no anonymous access)

16. Select **Create**

Cosmos DB

1. Open the [Azure Portal]()

2. Select **"+ Create a Resource"** and then enter **cosmos** in the search box

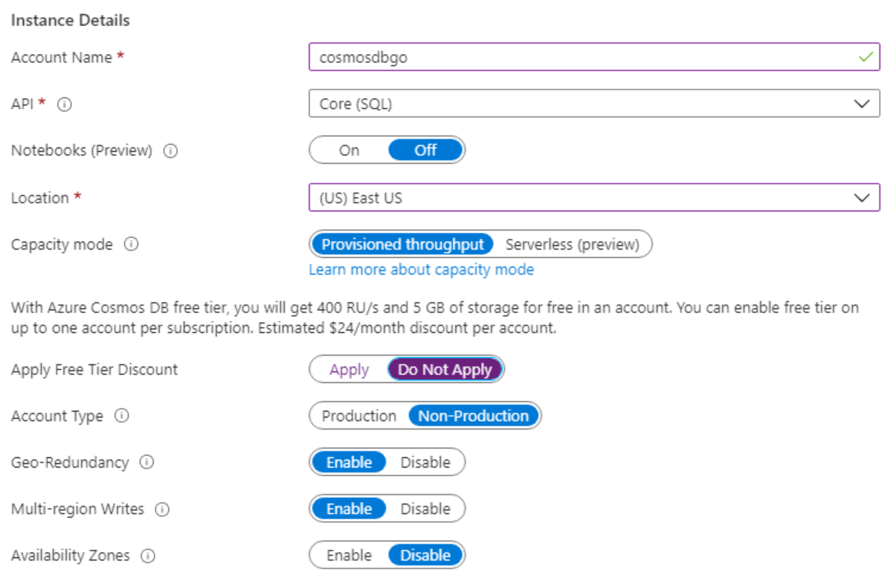
3. Choose **Azure Cosmos DB** from the available options.

4. Select **Create**

5. Select your subscription and resource group

6. Type a unique account name such as **ai100cosmosdbgo**

7. Select a location that matches your resource group



Instance Details

Account Name *	cosmosdbgo
API *	Core (SQL)
Notebooks (Preview)	On [✓ Off]
Location *	(US) East US
Capacity mode	[✓ Provisioned throughput] Serverless (preview)

Learn more about capacity mode

With Azure Cosmos DB free tier, you will get 400 RU/s and 5 GB of storage for free in an account. You can enable free tier on up to one account per subscription. Estimated $24/month discount per account.

Apply Free Tier Discount	Apply [✓ Do Not Apply]
Account Type	Production [✓ Non-Production]
Geo-Redundancy	[✓ Enable] Disable
Multi-region Writes	[✓ Enable] Disable
Availability Zones	Enable [✓ Disable]

8. Configure the remaining options as depicted in the above image

9. Select **Review + create**

10. Select **Create**

11. Navigate to the new resource, under **Settings**, select **Keys**

12. Copy the **URI** and the **PRIMARY KEY** to your notepad

Bot Builder SDK

We will use the Bot Builder template for C# to create bots in this course.

Download the Bot Builder SDK

1. Open a browser window to Bot Builder SDK v4 Template for C# here

2. Select **Download**

3. Navigate to the download folder location and double-click on the install

4. Ensure that all versions of Visual Studio are selected and select **Install**. If prompted, select **End Tasks**.

5. Select **Close**. You should now have the bot templates added to your Visual Studio templates.

Bot Emulator

We will be developing a bot using the latest .NET SDK (v4). In order to do local testing, we'll need to download the Bot Framework Emulator.

Download the Bot Framework Emulator

You can download the v4 Bot Framework Emulator for testing your bot locally. The instructions for the rest of the labs will assume you've downloaded the v4 Emulator.

1. Download the emulator by going to this page and downloading the most recent version of the emulator that has the tag "4.6.0" or higher (select the "*-windows-setup.exe" file, if you are using windows).

   > **Note** The emulator installs to `"C:\Users\_your-username\AppData\Local\Programs\@bfemulatormain\Bot Framework Emulator.exe"`, but you can gain access to it through the start menu by searching for **bot framework**.

Credits

Labs in this series were adapted from the Cognitive Services Tutorial and Learn AI BootCamp

Resources

To deepen your understanding of the architecture described here, and to involve your broader team in the development of AI solutions, we recommend reviewing the following resources:

- Cognitive Services - AI Engineer
- Cosmos DB - Data Engineer
- Azure Cognitive Search - Search Engineer
- Bot Developer Portal - AI Engineer

Next Steps

- Lab 02-01: Implement Computer Vision

Lab 2: Implement Computer Vision Capabilities for a Bot

Introduction

This hands-on lab guides you through creating an intelligent console application from end-to-end using Cognitive Services (specifically the Computer Vision API). We use the ImageProcessing portable class library (PCL), discussing its contents and how to use it in your own applications.

Next Steps

- Lab 02-02: Implement Computer Vision

Lab 2 - Implement Computer Vision

Introduction

We're going to build an end-to-end application that allows you to pull in your own pictures, use Cognitive Services to obtain a caption and some tags about the images. In later labs, we will build a Bot Framework bot using LUIS to allow easy, targeted querying of such images.

Lab 2.0: Objectives

In this lab, you will:

- Learn about the various Cognitive Services APIs
- Understand how to configure your apps to call Cognitive Services
- Build an application that calls various Cognitive Services APIs (specifically Computer Vision) in .NET applications

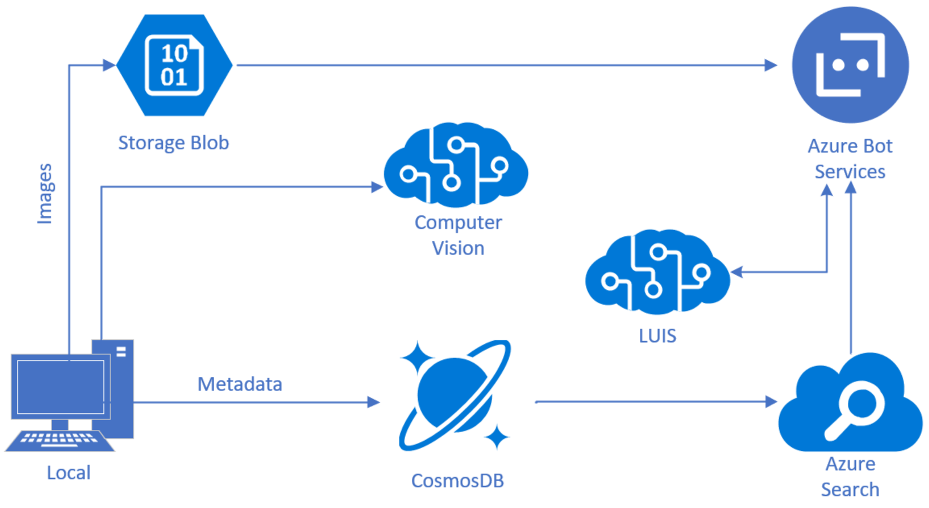
While there is a focus on Cognitive Services, you will also leverage Visual Studio 2019.

> **Note** if you have not already, follow the directions for creating your Azure account, Cognitive Services, and getting your api keys in Lab1-Technical_Requirements.md.

Lab 2.1: Architecture

We will build a simple C# application that allows you to ingest pictures from your local drive, then invoke the Computer Vision API to analyze the images and obtain tags and a description.

In the continuation of this lab throughout the course, we'll show you how to query your data, and then build a Bot Framework bot to query it. Finally, we'll extend this bot with LUIS to automatically derive intent from your queries and use those to direct your searches intelligently.



Lab 2.2: Resources

There are some directories in the main github repo folder:

- **sample_images**: Some sample images to use in testing your implementation of Cognitive Services.
- **code**: In here, there are two directories. Each folder contains a solution (.sln) that has several different projects for the lab.
- **Starter**: A starter project, which you can use if you want to learn about creating the code that is used in the project.
- **Finished**: A finished project that you will make use of to implement computer vision and work with the images in this lab.

Lab 2.3: Image Processing

Cognitive Services

Cognitive Services can be used to infuse your apps, websites and bots with algorithms to see, hear, speak, understand, and interpret your user needs through natural methods of communication.

There are five main categories for the available Cognitive Services:

- **Vision**: Image-processing algorithms to identify, caption and moderate your pictures
- **Knowledge**: Map complex information and data in order to solve tasks such as intelligent recommendations and semantic search
- **Language**: Allow your apps to process natural language with pre-built scripts, evaluate sentiment and learn how to recognize what users want
- **Speech**: Convert spoken audio into text, use voice for verification, or add speaker recognition to your app
- **Search**: Add Bing Search APIs to your apps and harness the ability to comb billions of webpages, images, videos, and news with a single API call

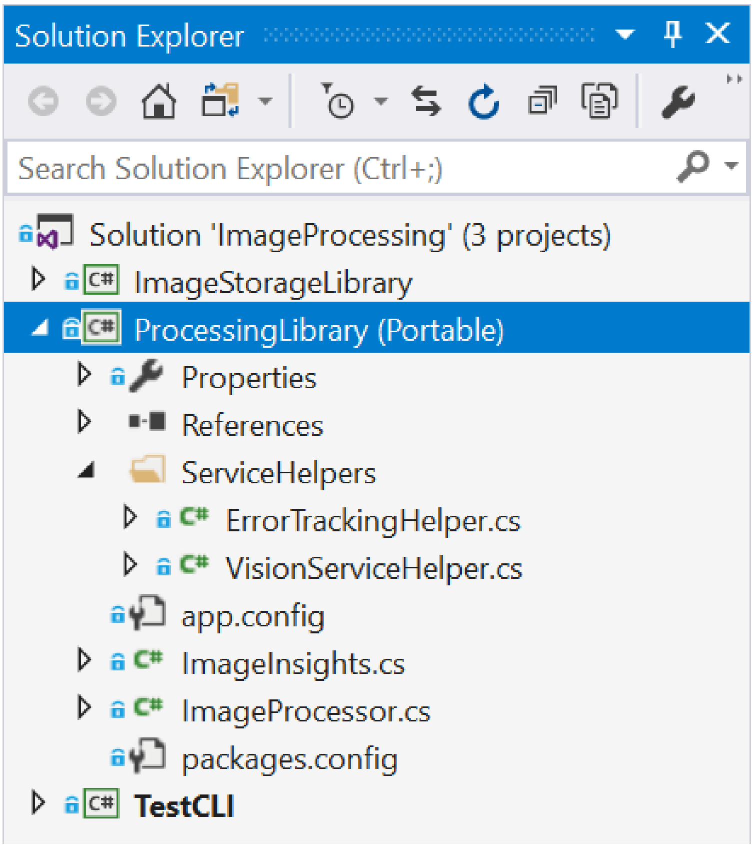
You can browse all of the specific APIs in the Services Directory.

Let's talk about how we're going to call Cognitive Services in our application by reviewing the sample code in the Finished project.

Image Processing Library

1. Open the **code/Finished/ImageProcessing.sln** solution

2. Within your `ImageProcessing` solution you'll find the `ProcessingLibrary` project. 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.



3. You should be able to pick up this portable class library and drop it in your other projects that involve Cognitive Services (some modification will be required depending on which Cognitive Services you want to use).

ProcessingLibrary: Service Helpers

1. 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.

2. 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

1. In the **ProcessingLibrary** project, navigate to the **ImageInsights.cs** file.

2. You can see that there are properties for `Caption` and `Tags` from the images, as well as a unique `ImageId`. "ImageInsights" collects the information from the Computer Vision API.

3. 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 `ImageProcessor.cs`to understand how it is being used. 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.).

Lab 2.4: Review ImageProcessor.cs

1. Navigate to **ImageProcessor.cs** within `ProcessingLibrary`.

2. Note the following `using` directives **to the top** of the class, above the namespace:

   ```
   csharp using System; using System.IO; using System.Linq; using
   System.Threading.Tasks; using Microsoft.ProjectOxford.Vision; using
   ServiceHelpers;
   ```

   Project Oxford was the project where many Cognitive Services got their start. As you can see, the NuGet Packages were even labeled under Project Oxford. In this scenario, we'll call `Microsoft.ProjectOxford.Vision` for the Computer Vision API. Additionally, we'll reference our service helpers (remember, these will make our lives easier). You'll have to reference different packages depending on which Cognitive Services you're leveraging in your application.

3. In **ImageProcessor.cs** we start by utliziling a method to process the image, `ProcessImageAsync`. The code will utilize asynchronous processing because it will utilize services to perform the actions.

   ```csharp public static async Task ProcessImageAsync(string imgPath, string imageId) { // Set up an array that we'll fill in over the course of the processor: VisualFeature[] DefaultVisualFeaturesList = new VisualFeature[] { VisualFeature.Tags, VisualFeature.Description };

   // Call the Computer Vision service and store the results in imageAnalysisResult: var imageAnalysisResult = await VisionServiceHelper.AnalyzeImageAsync(imgPath, DefaultVisualFeaturesList);

   // Create an entry in ImageInsights: ImageInsights result = new ImageInsights { ImageId = imageId, Caption = imageAnalysisResult.Description.Captions[0].Text, Tags = imageAnalysisResult.Tags.Select(t => t.Name).ToArray() };

   // Return results: return result; } ```

   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 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`.

4. Next, we want to call the Cognitive Service (specifically Computer Vision) and put the results in `imageAnalysisResult`.
   ```

5. We 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`). You will use the AnalyzeImageAsync method in order to return the visual features.

   ```
   csharp var imageAnalysisResult = await
   VisionServiceHelper.AnalyzeImageAsync(imgPath,
   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`).

6. The following code below accomplishes this.

   ```
   csharp ImageInsights result = new ImageInsights { ImageId = imageId,
   Caption = imageAnalysisResult.Description.Captions[0].Text, Tags =
   imageAnalysisResult.Tags.Select(t => t.Name).ToArray() };
   ```

   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".

7. Lastly, we need to close out the method by using the following line at the end of the method:

   ```
   csharp return result;
   ```

8. In order to use this application, we need to build the project, press **Ctrl-Shift-B**, of select the **Build** menu and choose **Build Solution**.

9. Work with your instructor to fix any errors.

Exploring Cosmos DB

Azure Cosmos DB is Microsoft's 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 can be quickly integrated into Azure Cognitive Search (which we'll do in a later lab).

Lab 2.5 (optional): Understanding CosmosDBHelper

Cosmos DB is not a focus of this lab, but if you're interested in what's going on - here are some highlights from the code we will be using:

1. Navigate to the `CosmosDBHelper.cs` class in the `ImageStorageLibrary` project. Review the code and the comments. Many of the implementations used can be found in the Getting Started guide.

2. Go to the `TestCLI` project's `Util.cs` file and review the `ImageMetadata` class (code and comments). This is where we turn the `ImageInsights` we retrieve from Cognitive Services into appropriate Metadata to be stored into Cosmos DB.

    - Finally, look in `Program.cs` in `TestCLI` and at `ProcessDirectoryAsync`. First, we check if the image and metadata have already been uploaded - we can use `CosmosDBHelper` to find the document by ID and to return `null` if the document doesn't exist. Next, if we've set `forceUpdate` or the image hasn't been processed before, we'll call the Cognitive Services using `ImageProcessor` from the `ProcessingLibrary` and retrieve the `ImageInsights`, which we add to our current `ImageMetadata`.

    - Once all of that is complete, we can store our image - first the actual image into Blob Storage using our `BlobStorageHelper` instance, and then the `ImageMetadata` into Cosmos DB using our `CosmosDBHelper` instance. If the document already existed (based on our previous check), we should update the existing document. Otherwise, we should be creating a new one.

Lab 2.6: 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.
1. In the **TestCLI** project, open the **settings.json** file
1. Add your specific environment settings from [Lab1-Technical_Requirements.md]()

```
> **Note** the url for cognitive services should end with
**/vision/v1.0** for the project oxford apis.  For example
'ENDPOINT/vision/v1.0`.
> ENDPOINT url can be found in your created Cognitive Services blade
"RESOURCE MANAGEMENT" "Keys and EndPoint"
```

1. If you have not already done so, build the project

2. Open a command prompt and navigate to the build directory for the **TestCLI** project. It should something like **{GitHubDir}\Lab2-Implement_Computer_Vision\code\Finished\TestCLI**.

   NOTE Do not navigate to the debug directory
   NOTE .net core 3.1 is requred installation can be find here https://dotnet.microsoft.com/download/dotnet-core/3.1

3. Run command **dotnet run**

   ```
   cmd 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
   ```

4. 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:

   ```
   cmd dotnet run -process "<%GitHubDir%>\AI-100-Design-Implement-
   Azure-AISol\Lab2-Implement_Computer_Vision\sample_images"
   ```

   Note Replace the <%GitHubDir%> value with the folder where you cloned the repository. Once it's done processing, you can query against your Cosmos DB directly using *TestCLI* as follows:
   `cmd dotnet run -query "select * from images"`

5. Take some time to look through the sample images (you can find them in /sample_images) and compare the images to the results in your application.

   Note You can also browse the results in the CosmosDb resource in Azure. Open the resource, then select **Data Explorer**. Expand the **metadata**

database, then select the **items** node. You will see several json documents that contains your results.

Credits

This lab was modified from this Cognitive Services Tutorial.

Resources

- Computer Vision API
- Bot Framework
- Services Directory
- Portable Class Library (PCL)
- Intelligent Kiosk sample application

Next Steps

- Lab 03-01: Basic Filter Bot Note! If you want to use the finished solution, you must fill in the following with keys in `settings.json` under TestCLI

```
json { "CognitiveServicesKeys": { "Vision": "VisionKeyHere" },
"AzureStorage": { "ConnectionString": "ConnectionStringHere",
"BlobContainer": "images" }, "CosmosDB": { "EndpointURI":
"CosmosURIHere", "Key": "CosmosKeyHere", "DatabaseName": "images",
"CollectionName": "metadata" } }
```

How To Run Finished Project.

1. Before you start you must complete Lab 1: Meeting the Technical Requirements. As result you should have Cosmos DB, Storage Account and Cognitive Service (General or Vision) deployed in your Azure subscription.

2. Modify `settings.json` in the `TestCLI` folder and provide following settings:

   ```
   JSON { "CognitiveServicesKeys": { "Url":
   "https://eastus.api.cognitive.microsoft.com/vision/v1.0", "Key":
   "01234567890" }, "AzureStorage": { "ConnectionString":
   "DefaultEndpointsProtocol=https;AccountName=yourdemoaccount;AccountK
   ey=yourdemokey;EndpointSuffix=core.windows.net", "BlobContainer":
   "images" }, "CosmosDB": { "EndpointURI":
   "https://yourdemodb.documents.azure.com:443/", "Key": "0123456789",
   "DatabaseName": "images", "CollectionName": "metadata" } }
   ```

   > NOTE The settings must be retrieved from existed resources.

   Following links should help you retrieve settings.

   - Get connections string from storage account
   - Get connection key for Cosmos DB
   - Get Key and endpoint URL for Cognitive Service

VS Code

1. Install C# extension

2. Open folder "Finished"

3. In the cmd console run command **dotnet build** in each of the folders: `ImageStorageLibrary`, `ProcessingLibrary`, `TestCLI`

4. From `TestCLI` folder run following commands:

   ```
   cmd dotnet run -process "<%GitHubDir%>\AI-100-Design-Implement-
   Azure-AISol\Lab2-Implement_Computer_Vision\sample_images"
   ```

   > **Note** Replace the <%GitHubDir%> value with the folder where you cloned the repository. Once it's done processing, you can query against your Cosmos DB directly using *TestCLI* as follows:

   ```
   cmd dotnet run -query "select * from images"
   ```

5. Take some time to look through the sample images (you can find them in /sample_images) and compare the images to the results in your application.

   > **Note** You can also browse the results in the CosmosDb resource in Azure. Open the resource, then select **Data Explorer**. Expand the **metadata**

> database, then select the **items** node. You will see several json documents that contains your results.

VS 2019

1. Open `ImageProcessing.sln` file in the root folder of projects.
2. Set `TestCLI` project as `Start up Project`.
3. Open `TestCLI` project properties. Select `Debug` tab and provide following string in `Application Arguments`:

   ```
   -process "<%GitHubDir%>\AI-100-Design-Implement-Azure-AISol\Lab2-
   Implement_Computer_Vision\sample_images"
   ```

   > **Note** Replace the <%GitHubDir%> value with the folder where you cloned the repository.

4. Run project by F5.
5. Take some time to look through the sample images (you can find them in /sample_images) and compare the images to the results in your application.

   > **Note** You can also browse the results in the CosmosDb resource in Azure. Open the resource, then select **Data Explorer**. Expand the **metadata** database, then select the **items** node. You will see several json documents that contains your results.

Sample Images

This directory contains `.jpg` images used for the Application.

Click here to view the directory on github.

Lab 3: Basic Filtering Bot

Introduction

In this lab, we will be setting up an intelligent bot from end-to-end that can respond to a user's chat window text prompt. We will be building on what we have already learned about building bots within Azure, but adding in a layer of custom logic to give our bot more bespoke functionality.

This bot will be built in the Microsoft Bot Framework. We will evoke the architecture that enables the bot interface to receive and respond with textual messages, and we will build logic that enables our bot to respond to inquiries containing specific text.

We will also be testing our bot in the Bot Emulator, and addressing the middleware that enables us to perform specialized tasks on the message that the bot receives from the user.

We will evoke some concepts pertaining to Azure Cognitive Search, and Microsoft's Language Understanding Intelligent Service (LUIS), but will not implement them in this lab.

Next Steps

- Lab 03-02: Basic Filter Bot

Lab 3: Creating a Basic filtering bot

Introduction

Every new technology brings with it as many opportunities as questions, and AI-powered technology has its own unique considerations. Be mindful of the following AI Ethics principles when designing and implementing your AI tools:

1. *Fairness*: Maximize efficiencies without destroying dignity
2. *Accountability*: AI must be accountable for its algorithm
3. *Transparency*: Guard against bias and destruction of human dignity
4. *Ethical Application*: AI must assist humanity and be designed for intelligent privacy

We encourage you to read more about the Ethical considerations when building intelligent apps.

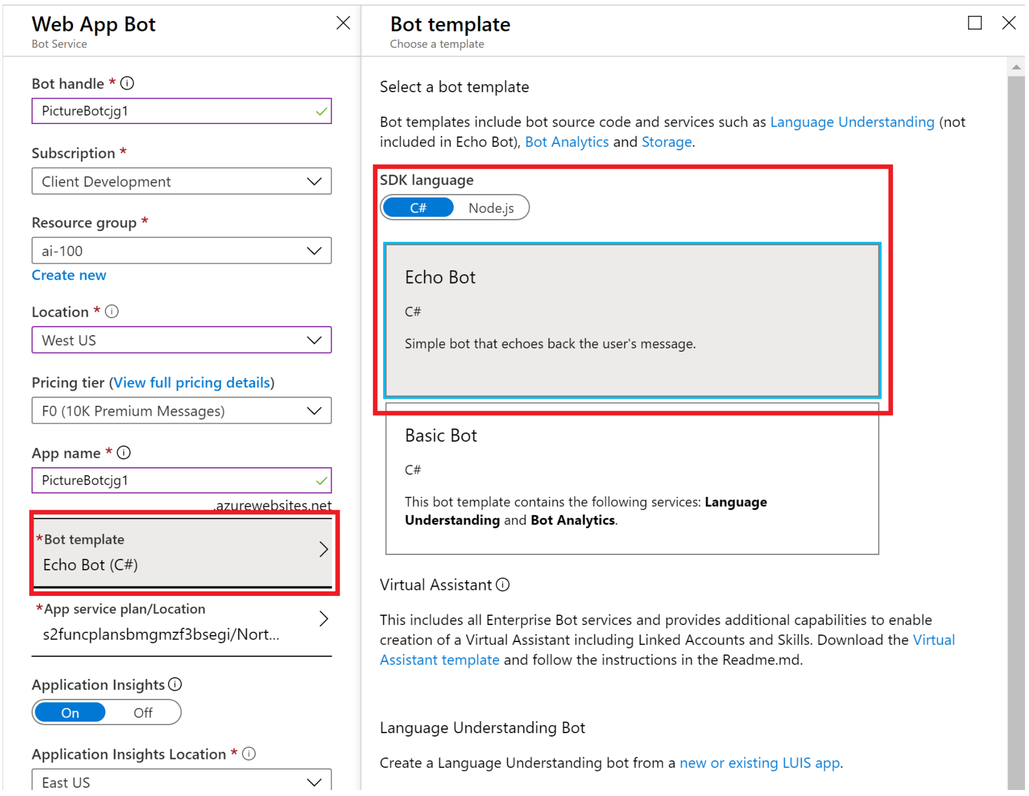
Pre-requisites

1. Follow the directions provided in Lab1-Technical_Requirements.md to download the v4 Bot Framework Emulator to enable you to test your bot locally.

Lab 3.0 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.

1. Navigate to the Azure portal.
2. In the portal, navigate to your resource group, then select **+Add** and search for **bot**.
3. Select **Web App Bot**, and select **Create**.
4. For the name, you'll have to create a unique identifier. We 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).
5. Select a region
6. For pricing tier, select **F0 (10K Premium Message)**.
7. Select the Bot template area
8. Select **C#**, then select **Echo Bot**, later we will update it to our our PictureBot.



Web App Bot
Bot Service

Bot handle * PictureBotcjg1
Subscription * Client Development
Resource group * ai-100
Create new
Location * West US
Pricing tier (View full pricing details) F0 (10K Premium Messages)
App name * PictureBotcjg1 .azurewebsites.net
*Bot template Echo Bot (C#)
*App service plan/Location s2funcplansbmgmzf3bsegi/Nort...
Application Insights On Off
Application Insights Location * East US

Bot template
Choose a template

Select a bot template

Bot templates include bot source code and services such as Language Understanding (not included in Echo Bot), Bot Analytics and Storage.

SDK language C# Node.js

Echo Bot
C#
Simple bot that echoes back the user's message.

Basic Bot
C#
This bot template contains the following services: **Language Understanding** and **Bot Analytics**.

Virtual Assistant

This includes all Enterprise Bot services and provides additional capabilities to enable creation of a Virtual Assistant including Linked Accounts and Skills. Download the Virtual Assistant template and follow the instructions in the Readme.md.

Language Understanding Bot

Create a Language Understanding bot from a new or existing LUIS app.

9. Select **OK**, make sure that **Echo Bot** is displayed.

10. Configure a new App service plan (put it in the same location as your bot)

11. You can choose to turn Application Insights on or off.

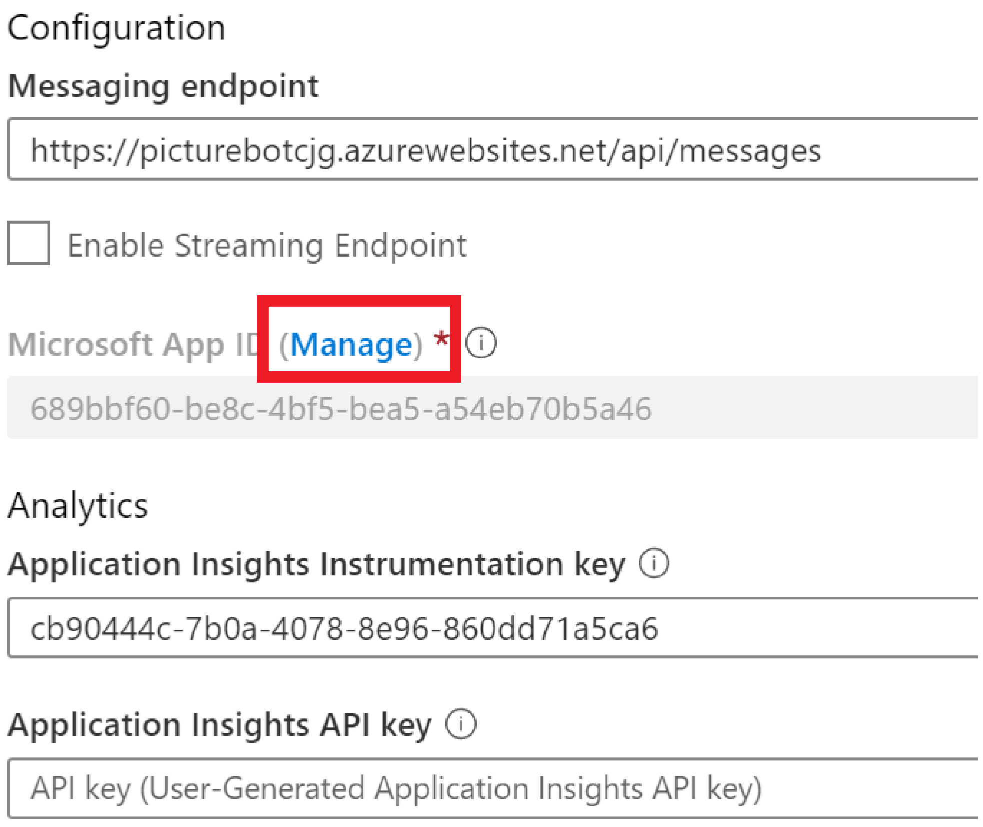
12. **Do not** change or select on **Auto create App ID and password**, we will get to that later.

13. Select **Create**

14. When it's deployed, navigate to the new Azure Web App Bot Resource.

15. Under **Bot Management**, select **Settings**

16. Select the **Manage** link for the **Microsoft App ID**


Configuration

Messaging endpoint

https://picturebotcjg.azurewebsites.net/api/messages

Enable Streaming Endpoint

Microsoft App ID (Manage) *

689bbf60-be8c-4bf5-bea5-a54eb70b5a46

Analytics

Application Insights Instrumentation key

cb90444c-7b0a-4078-8e96-860dd71a5ca6

Application Insights API key

API key (User-Generated Application Insights API key)

17. Select **New client secret**

18. For the name, type **PictureBot**

19. For the expires, select **Never**

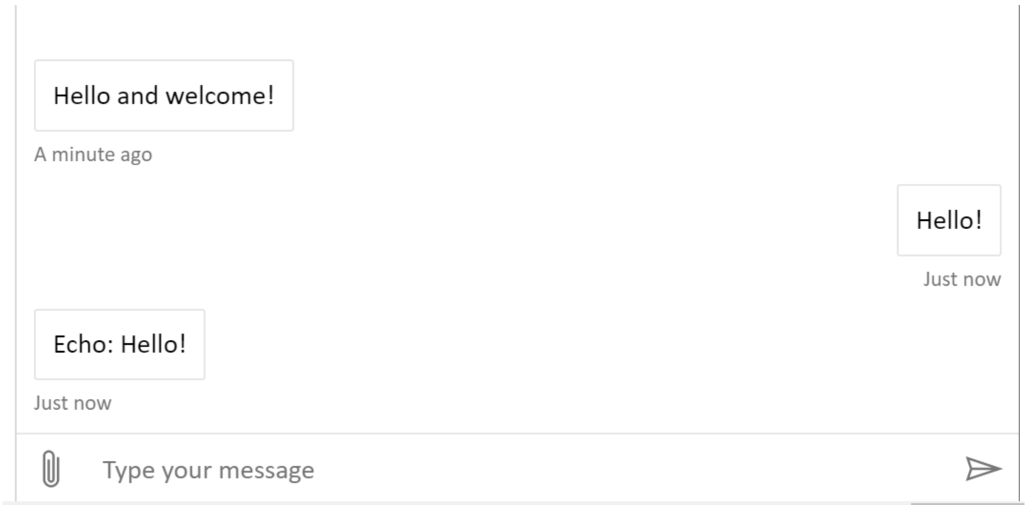
20. Select **Add**

21. Record the secret into notepad or similar for later use in the lab(s).

22. Select **Overview**, record the application id into notepad or similar for later use in the lab(s).

23. Navigate back to the **web app bot** resource, under **Bot management**, select the **Test in Web Chat** tab

24. Once it starts, explore what it is capable of doing. As you will see, it only echos back your message.



Hello and welcome!

A minute ago

Hello!

Just now

Echo: Hello!

Just now

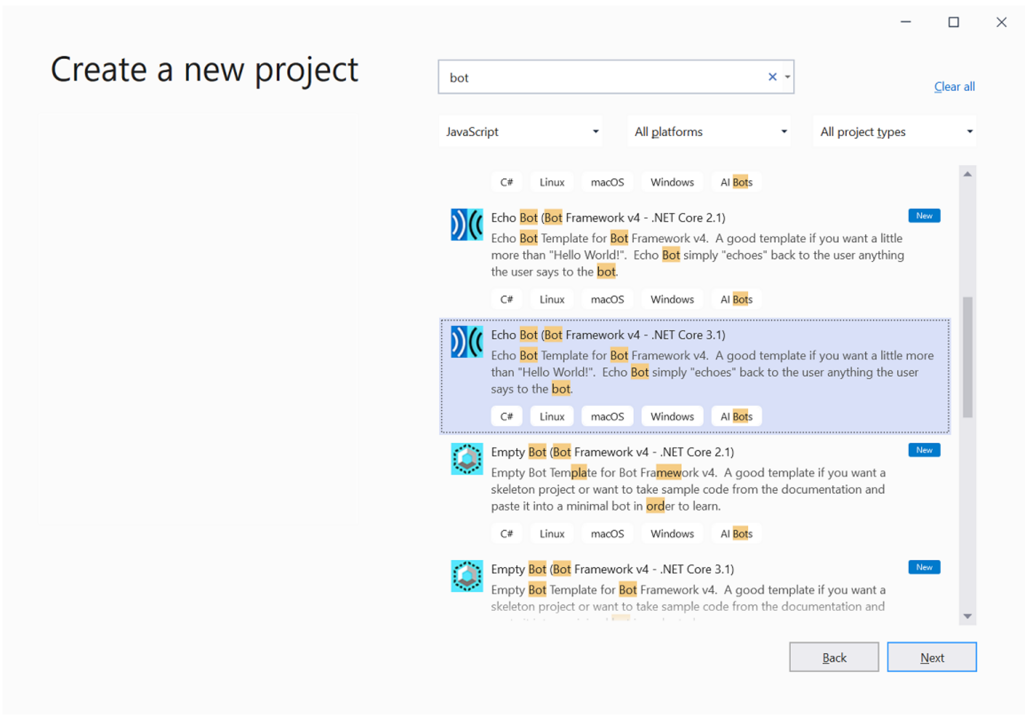
Lab 3.1: Creating a simple bot and running it

1. Open **Visual Studio 2019** or later

2. Select **Create new project**, search for **bot**.

3. Scroll down until you see **Echo Bot (Bot Framework v4)**

   > **[!CAUTION]** Depending on the version of Visual Studio installed, the below screenshot may be different from your own. If you see multiple versions listed for the Echo Bot template, choose **version 3.1** and not version 2.1.



Echo Bot (Bot Framework v4 - .NET Core 2.1)

Echo Bot Template for Bot Framework v4. A good template if you want a little more than "Hello World!". Echo Bot simply "echoes" back to the user anything the user says to the bot.

Echo Bot (Bot Framework v4 - .NET Core 3.1)

Echo Bot Template for Bot Framework v4. A good template if you want a little more than "Hello World!". Echo Bot simply "echoes" back to the user anything the user says to the bot.

Empty Bot (Bot Framework v4 - .NET Core 2.1)

Empty Bot Template for Bot Framework v4. A good template if you want a skeleton project or want to take sample code from the documentation and paste it into a minimal bot in order to learn.

Empty Bot (Bot Framework v4 - .NET Core 3.1)

Empty Bot Template for Bot Framework v4. A good template if you want a skeleton project or want to take sample code from the documentation and

4. Select **Next**

   > **Note** If you do not see the Echo Bot template, you need to install the Visual Studio add-in from the pre-req steps.

5. For the name, type **PictureBot**, select **Create**

6. Spend some time looking at all of the different things that are generated 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. You can find it and other useful shells and samples here.

7. Start by right-clicking on the Solution and selecting **Build**. This will restore the nuget packages.

8. Open the **appsettings.json** file, update it by adding your bot service information you recorded above:

   ```
   json { "MicrosoftAppId": "YOURAPPID", "MicrosoftAppPassword":
   "YOURAPPSECRET" }
   ```

9. As you may know, renaming a Visual Studio Solution/Project is a very sensitive task. **Carefully** complete the following tasks so all the names reflect PictureBot instead of EchoBot:

10. Right-click the **Bots/Echobot.cs** file, then select **Rename**, rename the class file to **PictureBot.cs**

11. If you are not prompted then you will need to manually rename the class and then change all references to the class to **PictureBot**. You will know if you missed one when you attempt to build the project.

12. Right-click the project, select **Manage Nuget Packages**

13. Select the **Browse** tab, and install the following packages, ensure that you are using latest version:

    - Microsoft.Bot.Builder.Azure.Blobs
    - Microsoft.Bot.Builder.Dialogs
    - Microsoft.Bot.Builder.AI.Luis
    - Microsoft.Bot.Builder.Integration.AspNet.Core
    - Azure.AI.TextAnalytics

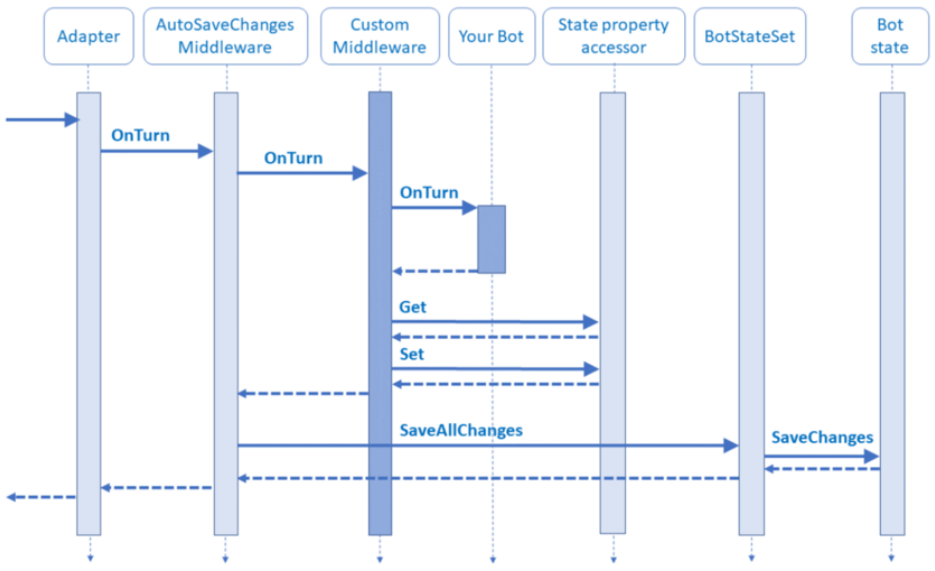
14. Build the solution.

    > **TIP**: If you only have one monitor and you would like to easily switch between instructions and Visual Studio, you can now add the instruction files to your Visual Studio solution by right-clicking on the project in Solution Explorer and selecting **Add > Existing Item**. Navigate to "Lab2," and add all the files of type "MD File."

Creating a Hello World bot

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 through the multiple layers of a bot application.



1. Open the **PictureBot.cs** file.

2. Review the `OnMessageActivityAsync` method with the code below. This method is called every turn of the conversation. You'll see later why that fact is important, but for now, remember that OnMessageActivityAsync is called on every turn.

3. Press **F5** to start debugging.

   A few things to **Note**

   - Your default.htm (under wwwroot) page will be displayed in a browser

   - Note the localhost port number for the web page. This should (and must) match the endpoint in your Emulator.

     Get stuck or broken? You can find the solution for the lab up until this point under {GitHubPath}/code/Finished/PictureBot-Part0. The readme file within the solution (once you open it) will tell you what keys you need to add in order to run the solution.

Using the Bot Framework Emulator

To interact with your bot:

- Launch the Bot Framework Emulator (note we are using the v4 Emulator). Select **Start**, then search for **Bot Emulator**.

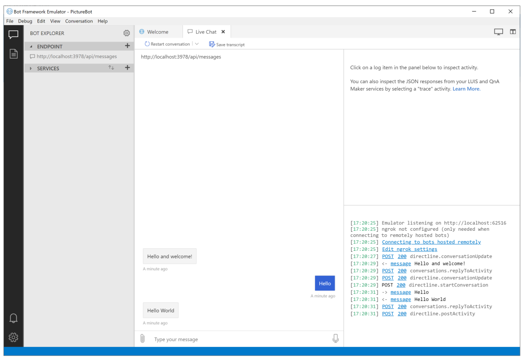
- On the welcome page, select **Create a new bot configuration**

- For the name, type **PictureBot**

- Enter the url that is displayed on your bot web page

- Enter the AppId and the App Secret your entered into the `appsettings.json`
- Select **Save and connect**, then save your .bot file locally
- You should now be able to converse with the bot.
- Type **hello**. The bot will respond with echoing your message similar to the Azure bot we created earlier.

Note You can select "Restart conversation" to clear the conversation history.



http://localhost:3978/api/messages

Click on a log item in the panel below to inspect activity.

You can also inspect the JSON responses from your LUIS and QnA Maker services by selecting a "trace" activity. Learn More.

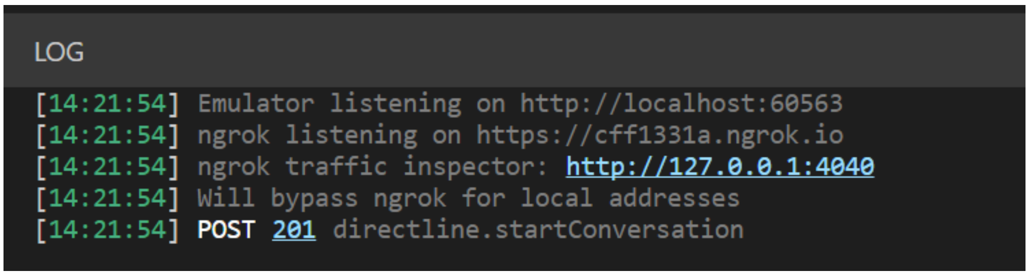
Hello and welcome!
A minute ago

Hello
A minute ago

Hello World
A minute ago

```
[17:20:25] Emulator listening on http://localhost:62516
[17:20:25] ngrok not configured (only needed when
connecting to remotely hosted bots)
[17:20:25] Connecting to bots hosted remotely
[17:20:25] Edit ngrok settings
[17:20:27] POST 200 directline.conversationUpdate
[17:20:29] <- message Hello and welcome!
[17:20:29] POST 200 conversations.replyToActivity
[17:20:29] POST 200 directline.conversationUpdate
[17:20:29] POST 200 directline.startConversation
[17:20:31] -> message Hello
[17:20:31] <- message Hello World
[17:20:31] POST 200 conversations.replyToActivity
[17:20:31] POST 200 directline.postActivity
```

In the Log, you should see something similar to the following:



LOG

```
[14:21:54] Emulator listening on http://localhost:60563
[14:21:54] ngrok listening on https://cff1331a.ngrok.io
[14:21:54] ngrok traffic inspector: http://127.0.0.1:4040
[14:21:54] Will bypass ngrok for local addresses
[14:21:54] POST 201 directline.startConversation
```

Note how it says we will bypass ngrok for local addresses. We will not be using ngrok in this workshop, but we would if we were connecting to our published version of the bot, we would do so via the 'production' endpoint. Open the 'production' endpoint and observe the difference between bots in different environments. This can be a useful feature when you're testing and comparing your development bot to your production bot.

You can read more about using the Emulator here.

1. Browse around and examine the sample bot code. In particular:
   - **Startup.cs**: is where we will add services/middleware and configure the HTTP request pipeline. There are many comments within to help you understand what is happening. Spend a few minutes reading through.
   - **PictureBot.cs**: The `OnMessageActivityAsync` method is the entry point which waits for a message from the user is where we can react to a message

once received and wait for further messages. We can use `turnContext.SendActivityAsync` to send a message from the bot back to the user.

Lab 3.2: Managing state and services

1. Navigate again to the **Startup.cs** file

2. Update the list of `using` statements by **adding** the following:

   ```csharp using System; using System.Linq; using System.Text.RegularExpressions; using Microsoft.Bot.Builder.Integration; using Microsoft.Bot.Configuration; using Microsoft.Bot.Connector.Authentication; using Microsoft.Extensions.Options; using Microsoft.Extensions.Logging; using PictureBot.Bots;

   using Microsoft.Bot.Builder.AI.Luis; using Microsoft.Bot.Builder.Dialogs; using Microsoft.Bot.Builder.Azure.Blobs; ```

   We won't use all of the above namespaces just yet, but can you guess when we might?

3. In the **Startup.cs** class, focus your attention on the `ConfigureServices` method which is used to add services to the bot. Review the contents carefully, noting what is built in for you.

   A few other notes for a deeper understanding:

   - If you're unfamiliar with dependency injection, you can read more about it here.
   - You can use local memory for this lab and testing purposes. For production, you'll have to implement a way to manage state data. In the big chunk of comments within `ConfigureServices`, there are some tips for this.
   - At the bottom of the method, you may notice we create and register state accessors. Managing state is a key in creating intelligent bots, which you can read more about here.

   Fortunately, this shell is pretty comprehensive, so we only have to add two items:

   - Middleware
   - Custom state accessors.

## Middleware

Middleware is simply a class or set of classes that sit between the adapter and your bot logic, and are added to your adapter's middleware collection during initialization.

The SDK allows you to write your own middleware or add reusable components of middleware created by others. Every activity coming in or out of your bot flows through your middleware. We'll get deeper into this later in the lab, but for now, it's important to understand that every activity flows through your middleware, because it
   ```

is located in the `ConfigureServices` method that gets called at run time (which runs in between every message being sent by a user and `OnMessageActivityAsync`).

1. Add a new folder called **Middleware**

2. Right-click on the **Middleware** folder and select **Add>Existing Item**.

3. Navigate to **{GitHubDir}\Lab3-Basic_Filter_Bot\code\Middleware**, select all three files, and select **Add**

4. Add the following variables to your **Startup** class:

   ```
   csharp private ILoggerFactory _loggerFactory;
   ```

5. Replace the following code in the **ConfigureServices** method:

   ```
   csharp services.AddTransient<IBot, PictureBot>();
   ```

   with the following code:

   ```csharp services.AddBot(options => { var appId = Configuration.GetSection("MicrosoftAppId")?.Value; var appSecret = Configuration.GetSection("MicrosoftAppPassword")?.Value;

   ```
               options.CredentialProvider = new
   SimpleCredentialProvider(appId, appSecret);

   // Creates a logger for the application to use.
   ILogger logger =
   _loggerFactory.CreateLogger<PictureBot.Bots.PictureBot>();

   // Catches any errors that occur during a conversation turn and logs
   them.
   options.OnTurnError = async (context, exception) =>
   {
       logger.LogError($"Exception caught : {exception}");
       await context.SendActivityAsync("Sorry, it looks like something
   went wrong.");
   };

   var middleware = options.Middleware;
   // Add middleware below with "middleware.Add(...."
   // Add Regex below
   ~~~

   }); ```

6. Replace the **Configure** method with the following code:

   ```csharp public void Configure(IApplicationBuilder app, ILoggerFactory loggerFactory) { _loggerFactory = loggerFactory;

   ~~~
   app.UseDefaultFiles()
               .UseBotFramework()
               .UseStaticFiles()
               .UseWebSockets()
   ```
   ```

```
                .UseRouting()
                .UseAuthorization()
                .UseEndpoints(endpoints =>
                {
                    endpoints.MapControllers();
                });
```

} ```

Custom state accessors

Before we talk about the custom state accessors that we need, it's important to back up. Dialogs, which we'll really get into in the next section, are an approach to implementing multi-turn conversation logic, which means they'll need to rely on a persisted state to know where in the conversation the users are. In our dialog-based bot, we'll use a DialogSet to hold the various dialogs. The DialogSet is created with a handle to an object called an "accessor".

In the SDK, an accessor implements the `IStatePropertyAccessor` interface, which basically means it provides the ability to get, set, and delete information regarding state, so we can keep track of which step a user is in a conversation.

1. For each accessor we create, we have to first give it a property name. For our scenario, we want to keep track of a few things:

    - `PictureState`

        - Have we greeted the user yet?
            - We don't want to greet them more than once, but we want to make sure we greet them at the beginning of a conversation.
        - Is the user currently searching for a specific term? If so, what is it?
            - We need to keep track of if the user has told us what they want to search for, and what it is they want to search for if they have.

    - `DialogState`

        - Is the user currently in the middle of a dialog?
            - This is what we'll use to determine where a user is in a given dialog or conversation flow. If you aren't familiar with dialogs, don't worry, we'll get to that soon.

    We can use these constructs to keep track of what we'll call `PictureState`.

2. In the **ConfigureServices** method of the **Startup.cs** file, add the `PictureState` within the list of custom state accessors and to keep track of the dialogs, you'll use the built-in `DialogState`:

    ```csharp // Create and register state accesssors. // Acessors created here are passed into the IBot-derived class on every turn. services.AddSingleton(sp => { var options = sp.GetRequiredService>().Value; if (options == null) { throw new
    ```

InvalidOperationException("BotFrameworkOptions must be configured prior to setting up the state accessors"); }

```
var conversationState = sp.GetRequiredService<ConversationState>();
//var conversationState =
services.BuildServiceProvider().GetService<ConversationState>();

if (conversationState == null)
{
    throw new InvalidOperationException("ConversationState must be
defined and added before adding conversation-scoped state
accessors.");
}

// Create the custom state accessor.
// State accessors enable other components to read and write
individual properties of state.
return new PictureBotAccessors(conversationState)
{
    PictureState = conversationState.CreateProperty<PictureState>
(PictureBotAccessors.PictureStateName),
    DialogStateAccessor =
conversationState.CreateProperty<DialogState>("DialogState"),
};
```

}); ```

3. You should see an error (red squiggly) beneath some of the terms. But before fixing them, you may be wondering why we had to create two accessors, why wasn't one enough?

    - `DialogState` is a specific accessor that comes from the `Microsoft.Bot.Builder.Dialogs` library. When a message is sent, the Dialog subsystem will call `CreateContext` on the `DialogSet`. Keeping track of this context requires the `DialogState` accessor specifically to get the appropriate dialog state JSON.

    - On the other hand, `PictureState` will be used to track particular conversation properties that we specify throughout the conversation (e.g. Have we greeted the user yet?)

    Don't worry about the dialog jargon yet, but the process should make sense. If you're confused, you can dive deeper into how state works.

4. Now back to the errors you're seeing. You've said you're going to store this information, but you haven't yet specified where or how. We have to update "PictureState.cs" and "PictureBotAccessor.cs" to have and access the information we want to store.

5. Right-click the project and select **Add->Class**, select a Class file and name it **PictureState**

6. Copy the following code into **PictureState.cs**.

```csharp using System.Collections.Generic;

namespace Microsoft.PictureBot { ///

/// Stores counter state for the conversation. /// Stored in and /// backed by . /// public class PictureState { ///
/// Gets or sets the number of turns in the conversation. ///
/// The number of turns in the conversation. public string Greeted { get; set; } = "not greeted"; public string Search { get; set; } = ""; public string Searching { get; set; } = "no"; } } ```

7. Review the code. This is where we'll store information about the active conversation. Feel free to add some comments explaining the purposes of the strings. Now that you have PictureState appropriately initialized, you can create the PictureBotAccessor, to remove the errors you were getting in **Startup.cs**.

8. Right-click the project and select **Add->Class**, select a Class file and name it **PictureBotAccessors**

9. Copy the following into it:

   ```csharp using System; using Microsoft.Bot.Builder; using Microsoft.Bot.Builder.Dialogs;

   namespace Microsoft.PictureBot { ///

   /// This class is created as a Singleton and passed into the IBot-derived constructor. /// - See constructor for how that is injected. /// - See the Startup.cs file for more details on creating the Singleton that gets /// injected into the constructor. ///
   public class PictureBotAccessors { ///
   /// Initializes a new instance of the class. /// Contains the and associated . ///
   /// The state object that stores the counter. public PictureBotAccessors(ConversationState conversationState) { ConversationState = conversationState ?? throw new ArgumentNullException(nameof(conversationState)); }

```
    /// <summary>
    /// Gets the <see cref="IStatePropertyAccessor{T}"/> name used
for the <see cref="CounterState"/> accessor.
    /// </summary>
    /// <remarks>Accessors require a unique name.</remarks>
    /// <value>The accessor name for the counter accessor.</value>
    public static string PictureStateName { get; } = $"
{nameof(PictureBotAccessors)}.PictureState";

    /// <summary>
    /// Gets or sets the <see cref="IStatePropertyAccessor{T}"/> for
CounterState.
    /// </summary>
    /// <value>
    /// The accessor stores the turn count for the conversation.
    /// </value>
```

```
    public IStatePropertyAccessor<PictureState> PictureState { get;
set; }

    /// <summary>
    /// Gets the <see cref="ConversationState"/> object for the
conversation.
    /// </summary>
    /// <value>The <see cref="ConversationState"/> object.</value>
    public ConversationState ConversationState { get; }

    /// <summary> Gets the IStatePropertyAccessor{T} name used for
the DialogState accessor. </summary>
    public static string DialogStateName { get; } = $"
{nameof(PictureBotAccessors)}.DialogState";

    /// <summary> Gets or sets the IStatePropertyAccessor{T} for
DialogState. </summary>
    public IStatePropertyAccessor<DialogState> DialogStateAccessor {
get; set; }
}
```

} ```

10. Review the code, notice the implementation of `PictureStateName` and `PictureState`.

11. Wondering if you configured it correctly? Return to **Startup.cs** and confirm your errors around creating the custom state accessors have been resolved.

Lab 3.3: Organizing code for bots

There are many different methods and preferences for developing bots. The SDK allows you to organize your code in whatever way you want. In these labs, we'll organize our conversations into different dialogs, and we'll explore a MVVM style of organizing code around conversations.

This PictureBot will be organized in the following way:

- **Dialogs** - the business logic for editing the models
- **Responses** - classes which define the outputs to the users
- **Models** - the objects to be modified

Create two new folders "**Responses**" and "**Models**" within your project by right-clicking the project and selecting **Add->New Folder**

Dialogs

You may already be familiar with Dialogs and how they work. If you aren't, read this page on Dialogs before continuing.

When a bot has the ability to perform multiple tasks, it is nice to be able to have multiple dialogs, or a set of dialogs, to help users navigate through different conversation flows. For our PictureBot, we want our users to be able to go through an initial menu flow, often referred to as a main dialog, and then branch off to different dialogs depending what they are trying to do - search pictures, share pictures, order pictures, or get help. We can do this easily by using a dialog container or what's referred to here as a `DialogSet`. Read about creating modular bot logic and complex dialogs before continuing.

For the purposes of this lab, we are going to keep things fairly simple, but after, you should be able to create a dialog set with many dialogs. For our PictureBot, we'll have two main dialogs:

- **MainDialog** - The default dialog the bot starts out with. This dialog will start other dialog(s) as the user requests them. This dialog, as it is the main dialog for the dialog set, will be responsible for creating the dialog container and redirecting users to other dialogs as needed.

- **SearchDialog* - *A dialog which manages processing search requests and returning those results to the user.*** Note* *We will evoke this functionality, but will not implement Search in this workshop.*

Since we only have two dialogs, we can keep it simple and put them in the PictureBot class. However, complex scenarios may require splitting them out into different dialogs in a folder (similar to how we'll separate Responses and Models).

1. Navigate back to **PictureBot.cs** and replace your `using` statements with the following:

   ```
   csharp using System.Collections.Generic; using System.Threading;
   using System.Threading.Tasks; using Microsoft.Bot.Builder; using
   Microsoft.Bot.Builder.Dialogs; using Microsoft.Bot.Schema; using
   Microsoft.PictureBot; using PictureBot.Responses;
   ```

You've just added access to your Models/Responses, as well as to the services LUIS and Azure Cognitive Search. Finally, the Newtonsoft references will help you parse the responses from LUIS, which we will see in a subsequent lab.

Next, we'll need to override the `OnTurnAsync` method with one that processes incoming messages and then routes them through the various dialogs.

1. Replace the **PictureBot** class with the following:

   ```csharp ///

   /// Represents a bot that processes incoming activities. /// For each user interaction, an instance of this class is created and the OnTurnAsync method is called. /// This is a Transient lifetime service. Transient lifetime services are created /// each time they're requested. For each Activity received, a new instance of this /// class is created. Objects that are expensive to construct, or have a lifetime /// beyond the single turn, should be carefully managed. /// For example, the object and associated /// object are created with a singleton lifetime. ///
   /// ///
   Contains the set of dialogs and prompts for the picture bot.
   public class PictureBot : ActivityHandler { private readonly PictureBotAccessors _accessors; // Initialize LUIS Recognizer

   ```
   private readonly ILogger _logger;
   private DialogSet _dialogs;

   /// <summary>
   /// Every conversation turn for our PictureBot will call this
   method.
   /// There are no dialogs used, since it's "single turn" processing,
   meaning a single
   /// request and response. Later, when we add Dialogs, we'll have to
   navigate through this method.
   /// </summary>
   /// <param name="turnContext">A <see cref="ITurnContext"/>
   containing all the data needed
   /// for processing this conversation turn. </param>
   /// <param name="cancellationToken">(Optional) A <see
   cref="CancellationToken"/> that can be used by other objects
   /// or threads to receive notice of cancellation.</param>
   /// <returns>A <see cref="Task"/> that represents the work queued to
   execute.</returns>
   /// <seealso cref="BotStateSet"/>
   /// <seealso cref="ConversationState"/>
   /// <seealso cref="IMiddleware"/>
   public override async Task OnTurnAsync(ITurnContext turnContext,
   CancellationToken cancellationToken = default(CancellationToken))
   ```
   ```

```
{
    if (turnContext.Activity.Type is "message")
    {
        // Establish dialog context from the conversation state.
        var dc = await _dialogs.CreateContextAsync(turnContext);
        // Continue any current dialog.
        var results = await
dc.ContinueDialogAsync(cancellationToken);

        // Every turn sends a response, so if no response was sent,
        // then there no dialog is currently active.
        if (!turnContext.Responded)
        {
            // Start the main dialog
            await dc.BeginDialogAsync("mainDialog", null,
cancellationToken);
        }
    }
}
/// <summary>
/// Initializes a new instance of the <see cref="PictureBot"/>
class.
/// </summary>
/// <param name="accessors">A class containing <see
cref="IStatePropertyAccessor{T}"/> used to manage state.</param>
/// <param name="loggerFactory">A <see cref="ILoggerFactory"/> that
is hooked to the Azure App Service provider.</param>
/// <seealso cref="https://docs.microsoft.com/en-
us/aspnet/core/fundamentals/logging/?view=aspnetcore-2.1#windows-
eventlog-provider"/>
public PictureBot(PictureBotAccessors accessors, ILoggerFactory
loggerFactory /*, LuisRecognizer recognizer*/)
{
    if (loggerFactory == null)
    {
        throw new
System.ArgumentNullException(nameof(loggerFactory));
    }

    // Add instance of LUIS Recognizer

    _logger = loggerFactory.CreateLogger<PictureBot>();
    _logger.LogTrace("PictureBot turn start.");
    _accessors = accessors ?? throw new
System.ArgumentNullException(nameof(accessors));

    // The DialogSet needs a DialogState accessor, it will call it
when it has a turn context.
    _dialogs = new DialogSet(_accessors.DialogStateAccessor);

    // This array defines how the Waterfall will execute.
    // We can define the different dialogs and their steps here
    // allowing for overlap as needed. In this case, it's fairly
simple
    // but in more complex scenarios, you may want to separate out
the different
    // dialogs into different files.
    var main_waterfallsteps = new WaterfallStep[]
    {
        GreetingAsync,
        MainMenuAsync,
    };
    var search_waterfallsteps = new WaterfallStep[]
    {
        // Add SearchDialog water fall steps

    };

    // Add named dialogs to the DialogSet. These names are saved in
the dialog state.
    _dialogs.Add(new WaterfallDialog("mainDialog",
main_waterfallsteps));
    _dialogs.Add(new WaterfallDialog("searchDialog",
search_waterfallsteps));
    // The following line allows us to use a prompt within the
dialogs
    _dialogs.Add(new TextPrompt("searchPrompt"));
}
// Add MainDialog-related tasks

// Add SearchDialog-related tasks

// Add search related tasks
```

} ```

2. Spend some time reviewing and discussing this shell with a fellow workshop participant. You should understand the purpose of each line before continuing.

3. We'll add some more to this in a bit. You can ignore any errors for now.

Responses

So before we fill out our dialogs, we need to have some responses ready. Remember, we're going to keep dialogs and responses separate, because it results in cleaner code, and an easier way to follow the logic of the dialogs. If you don't agree or understand now, you will soon.

1. In the **Responses** folder, create two classes, called **MainResponses.cs** and **SearchResponses.cs**. As you may have figured out, the Responses files will simply contain the different outputs we may want to send to users, no logic.

2. Within **MainResponses.cs** replace the code with the following:

   ```csharp using System.Threading.Tasks; using Microsoft.Bot.Builder;

   namespace PictureBot.Responses { public class MainResponses { public static async Task ReplyWithGreeting(ITurnContext context) { await context.SendActivityAsync("Hello, Im a Picture Bot"); } public static async Task ReplyWithHelp(ITurnContext context) { await context.SendActivityAsync($"I can search for pictures, share pictures and order prints of pictures."); } public static async Task ReplyWithResumeTopic(ITurnContext context) { await context.SendActivityAsync($"What can I do for you?"); } public static async Task ReplyWithConfused(ITurnContext context) { // Add a response for the user
   ```

if Regex or LUIS doesn't know // What the user is trying to communicate await context.SendActivityAsync($"I'm sorry, I don't understand."); } public static async Task ReplyWithLuisScore(ITurnContext context, string key, double score) { await context.SendActivityAsync($"Intent: {key} ({score})."); } public static async Task ReplyWithShareConfirmation(ITurnContext context) { await context.SendActivityAsync($"Posting your picture(s) on twitter..."); } public static async Task ReplyWithOrderConfirmation(ITurnContext context) { await context.SendActivityAsync($"Ordering standard prints of your picture(s)..."); } public static async Task ReplyWithSearchConfirmation(ITurnContext context) { await context.SendActivityAsync($"Searching picture(s)..."); } } } ```

Note that there are two responses with no values (ReplyWithGreeting and ReplyWithConfused). Fill these in as you see fit.

3. Within "SearchResponses.cs" replace the code with the following:

   ```csharp using Microsoft.Bot.Builder; using System; using System.Collections.Generic; using System.Linq; using System.Threading.Tasks; using Microsoft.Bot.Schema;

   namespace PictureBot.Responses { public class SearchResponses { // add a task called "ReplyWithSearchRequest" // it should take in the context and ask the // user what they want to search for public static async Task ReplyWithSearchConfirmation(ITurnContext context, string utterance) { await context.SendActivityAsync($"Ok, searching for pictures of {utterance}"); } public static async Task ReplyWithNoResults(ITurnContext context, string utterance) { await context.SendActivityAsync("There were no results found for \"" + utterance + "\"."); } } } ```

4. Notice a whole task is missing. Fill in as you see fit, but make sure the new task has the name "ReplyWithSearchRequest", or you may have issues later.

## Models

Due to time limitations, we will not be walking through creating all the models. They are straightforward, but we recommend taking some time to review the code within after you've added them.

1. Right-click on the **Models** folder and select **Add>Existing Item**.

2. Navigate to **{GitHubDir}\Lab3-Basic_Filter_Bot\code\Models**, select all three files, and select **Add**.
   ```

Lab 3.4: Regex and Middleware

There are a number of things that we can do to improve our bot. First of all, we may not want to call LUIS for a simple "search pictures" message, which the bot will get fairly frequently from its users. A simple regular expression could match this, and save us time (due to network latency) and money (due to cost of calling the LUIS service).

Also, as the complexity of our bot grows and we are taking the user's input and using multiple services to interpret it, we need a process to manage that flow. For example, try regular expressions first, and if that doesn't match, call LUIS, and then perhaps we also drop down to try other services like QnA Maker or Azure Cognitive Search. A great way to manage this is through Middleware, and the SDK does a great job supporting that.

Before continuing with the lab, learn more about middleware and the Bot Framework SDK:

- Overview and Architecture
- Middleware
- Creating Middleware

Ultimately, we'll use some middleware to try to understand what users are saying with regular expressions (Regex) first, and if we can't, we'll call LUIS. If we still can't, then we'll drop down to a generic "I'm not sure what you mean" response, or whatever you put for "ReplyWithConfused."

1. In **Startup.cs**, below the "Add Regex below" comment within `ConfigureServices`, add the following:

   ```
   csharp middleware.Add(new RegExpRecognizerMiddleware() .AddIntent("search", new Regex("search picture(?:s)(.)|search pic(?:s)(.)", RegexOptions.IgnoreCase)) .AddIntent("share", new Regex("share picture(?:s)(.)|share pic(?:s)(.)", RegexOptions.IgnoreCase)) .AddIntent("order", new Regex("order picture(?:s)(.)|order print(?:s)(.)|order pic(?:s)(.)", RegexOptions.IgnoreCase)) .AddIntent("help", new Regex("help(.*)", RegexOptions.IgnoreCase)));
   ```

   > We're really just skimming the surface of using regular expressions. If you're interested, you can learn more here.

2. You may notice that the `options.State` has been deprecated. Let's migrate to the newest method:

3. Remove the following code:

```csharp var conversationState = new ConversationState(dataStore);

options.State.Add(conversationState); ```

4. Add following code in the bottom of the `ConfigureServices`

   ```csharp // Create the User state. services.AddSingleton(sp => { var dataStore = sp.GetRequiredService(); return new UserState(dataStore); });

   // Create the Conversation state. services.AddSingleton(sp => { var dataStore = sp.GetRequiredService(); return new ConversationState(dataStore); });

   // Create the IStorage. services.AddSingleton(sp => { return new MemoryStorage(); }); ```

5. Without adding LUIS, our bot is really only going to pick up on a few variations, but it should capture a good bit of messages, if the users are using the bot for searching and sharing and ordering pictures.

   > Aside: One might argue that the user shouldn't have to type "help" to get a menu of clear options on what the bot can do; rather, this should be the default experience on first contact with the bot. **Discoverability** is one of the biggest challenges for bots - letting the users know what the bot is capable of doing. Good bot design principles can help.

# Lab 3.5: Running the bot

## MainDialog, Again

Let's get down to business. We need to fill out MainDialog within PictureBot.cs so that our bot can react to what users say they want to do. Based on our results from Regex, we need to direct the conversation in the right direction. Read the code carefully to confirm you understand what it's doing.

1. In **PictureBot.cs**, add following namespaces to the top of the file

   ```
   csharp using Microsoft.Bot.Schema; using Microsoft.PictureBot; using PictureBot.Responses;
   ```

2. Add following line to the top of the class.

   ```
   csharp private readonly PictureBotAccessors _accessors;
   ```

3. Add the following method by pasting code:

   ```csharp // If we haven't greeted a user yet, we want to do that first, but for the rest of the // conversation we want to remember that we've already greeted them. private async Task GreetingAsync(WaterfallStepContext stepContext, CancellationToken cancellationToken) { // Get the state for the current step in the conversation var state = await _accessors.PictureState.GetAsync(stepContext.Context, () => new PictureState());

   ```
   // If we haven't greeted the user
   if (state.Greeted == "not greeted")
   {
       // Greet the user
       await MainResponses.ReplyWithGreeting(stepContext.Context);
       // Update the GreetedState to greeted
       state.Greeted = "greeted";
       // Save the new greeted state into the conversation state
       // This is to ensure in future turns we do not greet the user
   again
       await
   _accessors.ConversationState.SaveChangesAsync(stepContext.Context);
       // Ask the user what they want to do next
       await MainResponses.ReplyWithHelp(stepContext.Context);
       // Since we aren't explicitly prompting the user in this step,
   we'll end the dialog
       // When the user replies, since state is maintained, the else
   clause will move them
       // to the next waterfall step
       return await stepContext.EndDialogAsync();
   }
   else // We've already greeted the user
   {
       // Move to the next waterfall step, which is MainMenuAsync
   ```
   ```

```
    return await stepContext.NextAsync();
}
```

}

// This step routes the user to different dialogs // In this case, there's only one other dialog, so it is more simple, // but in more complex scenarios you can go off to other dialogs in a similar public async Task MainMenuAsync(WaterfallStepContext stepContext, CancellationToken cancellationToken) { // Check if we are currently processing a user's search var state = await _accessors.PictureState.GetAsync(stepContext.Context);

```
// If Regex picks up on anything, store it
var recognizedIntents =
stepContext.Context.TurnState.Get<IRecognizedIntents>();
// Based on the recognized intent, direct the conversation
switch (recognizedIntents.TopIntent?.Name)
{
    case "search":
        // switch to the search dialog
        return await stepContext.BeginDialogAsync("searchDialog",
null, cancellationToken);
    case "share":
        // respond that you're sharing the photo
        await
MainResponses.ReplyWithShareConfirmation(stepContext.Context);
        return await stepContext.EndDialogAsync();
    case "order":
        // respond that you're ordering
        await
MainResponses.ReplyWithOrderConfirmation(stepContext.Context);
        return await stepContext.EndDialogAsync();
    case "help":
        // show help
        await MainResponses.ReplyWithHelp(stepContext.Context);
        return await stepContext.EndDialogAsync();
    default:
        {
            await
MainResponses.ReplyWithConfused(stepContext.Context);
            return await stepContext.EndDialogAsync();
        }
}
```

} ```

4. Find function `OnMessageActivityAsync` and replace with new function `OnTurnAsync`

   ```csharp public override async Task OnTurnAsync(ITurnContext turnContext, CancellationToken cancellationToken = default(CancellationToken)) { if (turnContext.Activity.Type is "message") { // Establish dialog context from the conversation state. var dc = await _dialogs.CreateContextAsync(turnContext); // Continue any current dialog. var results = await dc.ContinueDialogAsync(cancellationToken);
   ```

```
    // Every turn sends a response, so if no response was sent,
    // then there no dialog is currently active.
    if (!turnContext.Responded)
    {
        // Start the main dialog
        await dc.BeginDialogAsync("mainDialog", null,
cancellationToken);
    }
}
```

} ```

5. Press **F5** to run the bot.

6. Using the bot emulator, test the bot by sending some commands:

   - help
   - share pics
   - order pics
   - search pics

   Note If you get a 500 error in your bot, you can place a break point in the **Startup.cs** file inside the **OnTurnError** delegate method. The most common error is a mismatch of the AppId and AppSecret.

7. If the only thing that didn't give you the expected result was "search pics", everything is working how you configured it. "search pics" failing is the expected behavior at this point in the lab, but why? Have an answer before you move on!

   Hint: Use break points to trace matching to case "search", starting from **PictureBot.cs**. Get stuck or broken? You can find the solution for the lab up until this point under resources/code/Finished. The readme file within the solution (once you open it) will tell you what keys you need to add in order to run the solution. We recommend using this as a reference, not as a solution to run, but if you choose to run it, be sure to add the necessary keys for your environment.

Resources

- Bot Builder Basics
- .NET Bot Builder SDK tutorial
- Bot Service Documentation
- Deploy your bot

Next Steps

- Lab 04-01: Log Chat You don't need to add any keys to run this solution.

In appsettings.json, you will need to set your Bot Service filepath and secret. You don't need to add any keys to run this solution.

In appsettings.json, you will need to set your Bot Service filepath and secret. ���# How To Run Finished PictureBot project

Bot Framework v4 echo bot sample.

This bot has been created using Bot Framework, it shows how to create a simple bot that accepts input from the user and echoes it back.

Prerequisites

- .NET Core SDK version 3.1

```
bash # determine dotnet version dotnet --version
```

- File `settings.json` must be updated according the Lab instruction. Example of configuration:

```
JSON { "MicrosoftAppId": "1234567890", "MicrosoftAppPassword":
"abcdefg", "BlobStorageConnectionString":
"DefaultEndpointsProtocol=https;AccountName=youraccount;AccountKey=yourk
ey;EndpointSuffix=core.windows.net", "BlobStorageContainer": "chatlog",
"luisAppId": "1234567890", "luisAppKey": "1234567890", "luisEndPoint":
"https://eastus.api.cognitive.microsoft.com/", "cogsBaseUrl":
"https://eastus.api.cognitive.microsoft.com/", "cogsKey": "1234567890" }
```
- useful links

```
* [Create new or pick existed AppID](https://docs.microsoft.com/en-
us/azure/bot-service/bot-service-resources-faq-azure?view=azure-bot-
service-4.0)
* [Get connections string from storage account]
(https://docs.microsoft.com/en-us/azure/storage/common/storage-account-
keys-manage?tabs=azure-portal#view-account-access-keys)
* [Get connection key for Cosmos DB](https://docs.microsoft.com/en-
us/azure/cosmos-db/create-sql-api-python#update-your-connection-string)
* [Get Key and endpoint URL for Cognitive Service]
(https://docs.microsoft.com/en-us/azure/search/search-create-service-
portal#get-a-key-and-url-endpoint)
* [Get key for LUIS service](https://docs.microsoft.com/en-
us/azure/cognitive-services/luis/luis-container-howto?tabs=v3#gathering-
required-parameters)
```

To try this sample

- In a terminal, navigate to `PictureBot`

  ```bash
  ```

change into project folder

cd # PictureBot ```

- Run the bot from a terminal or from Visual Studio, choose option A or B.

A) From a terminal

```
bash # run the bot dotnet run
```

B) Or from Visual Studio

- Launch Visual Studio
- File -> Open -> Project/Solution
- Navigate to `PictureBot` folder
- Select `PictureBot.csproj` file
- Press `F5` to run the project

Testing the bot using Bot Framework Emulator

Bot Framework Emulator is a desktop application that allows bot developers to test and debug their bots on localhost or running remotely through a tunnel.

- Install the Bot Framework Emulator version 4.5.0 or greater from here

Connect to the bot using Bot Framework Emulator

- Launch Bot Framework Emulator
- File -> Open Bot
- Enter a Bot URL of `http://localhost:3978/api/messages`
- Enter your AppID and Secret.

Deploy the bot to Azure

To learn more about deploying a bot to Azure, see Deploy your bot to Azure for a complete list of deployment instructions.

Further reading

- Bot Framework Documentation
- Bot Basics
- Activity processing
- Azure Bot Service Introduction
- Azure Bot Service Documentation
- .NET Core CLI tools
- Azure CLI
- Azure Portal
- Language Understanding using LUIS
- Channels and Bot Connector Service

Developing Intelligent Applications with LUIS and Azure Search

This hands-on lab guides you through creating an intelligent bot from end-to-end using the Microsoft Bot Framework, Azure Search, and Microsoft's Language Understanding Intelligent Service (LUIS).

> **[08/15/2018] Important Note!** The v4 SDK for the bot framework recently went public preview. It is unknown when it will go GA. If you would like to complete lab02.2-building_bots with the v4 SDK (instead of the v3 SDK), we have made that the default lab for 2.2. If you are using these materials as part of a class (i.e. not self-study), **defer to instructor guidance**. If you are an instructor redelivering this course and have questions, please email learnanalytics@microsoft.com.

Objectives

In this workshop, you will: - Understand how to implement Azure Search features to provide a positive search experience inside applications - Build an intelligent bot using Microsoft Bot Framework that leverages LUIS and Azure Search - Use Regular Expressions and Scorable Groups to make bots more efficient

Prerequisites

This workshop is meant for an AI Developer on Azure. Since this is a short workshop, there are certain things you need before you arrive.

Firstly, you should have experience with Visual Studio. We will be using it for everything we are building in the workshop, so you should be familiar with how to use it to create applications. Additionally, this is not a class where we teach you how to code or develop applications. We assume you know how to code in C# (you can learn here), but you do not know how to implement advanced Search and NLP (natural language processing) solutions.

Secondly, you should have some experience developing bots with Microsoft's Bot Framework. We won't spend a lot of time discussing how to design them or how dialogs work. If you are not familiar with the Bot Framework, you should take this Microsoft Virtual Academy course prior to attending the workshop.

Thirdly, you should have experience with the portal and be able to create resources (and spend money) on Azure. We will not be providing Azure passes for this workshop.

> **Note** This workshop was developed and tested on a Data Science Virtual Machine (DSVM) with Visual Studio Community Version 15.4.0

Introduction

We're going to build an end-to-end scenario that allows you to pull in your own pictures, use Cognitive Services to find objects and people in the images, figure out how those people are feeling, and store all of that data into a NoSQL Store (CosmosDB). We'll use that NoSQL Store to populate an Azure Search index, and then build a Bot Framework bot using LUIS to allow easy, targeted querying.

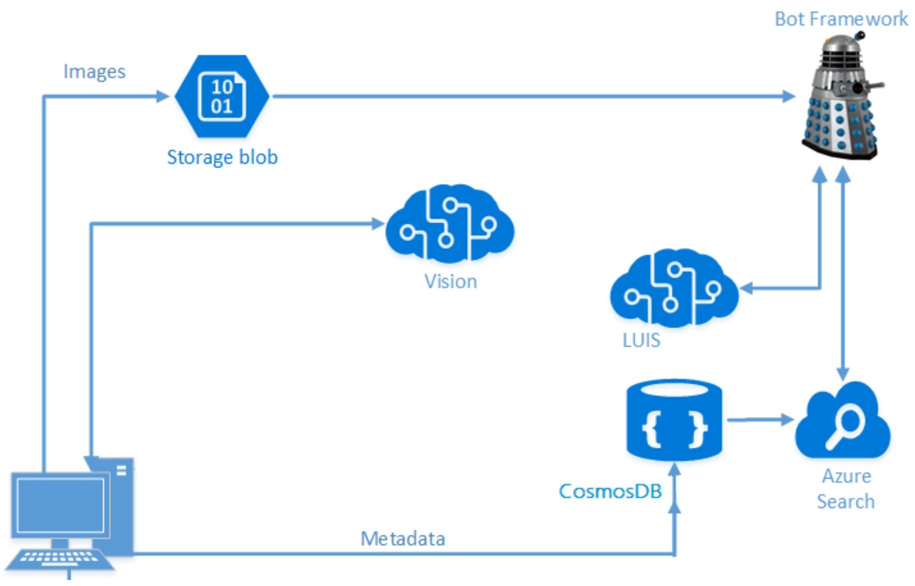
> **Note** This lab combines some of the results obtained from various labs (Computer Vision, Azure Search, and LUIS) from earlier in this workshop. If you did not complete the above listed labs, you will need to complete the Azure Search and LUIS lab before moving forwards. Alternatively, you can request to use a neighbor's keys from their Azure Search/LUIS labs.

Architecture

In a previous lab, we built a simple C# application that allows you to ingest pictures from your local drive, then invoke the Computer Vision Cognitive Service to grab tags and a description for those images.

Once we had this data, we processed it and stored all the information needed in CosmosDB, our NoSQL PaaS offering.

Once we had it in CosmosDB, we built an Azure Search Index on top of it. Next, we will build a Bot Framework bot to query it. We'll also extend this bot with LUIS to automatically derive intent from your queries and use those to direct your searches intelligently.



This lab was modified from this Cognitive Services Tutorial.

Navigating the GitHub

There are several directories in the resources folder:

- **assets**, **case**, **instructor**: You can ignore these folders for the purposes of this lab.
- **code**: In here, there are several directories that we will use:
  - **Models**: These classes will be used when we add search to our PictureBot.
  - **Finished-PictureBot_Regex**: Here there is the finished PictureBot.sln that includes additions for Regex. If you fall behind or get stuck, you can refer to this.
  - **Finished-PictureBot_Search**: Here there is the finished PictureBot.sln that includes additions for Regex and Search. If you fall behind or get stuck, you can refer to this.
  - **Finished-PictureBot_LUIS**: Here there is the finished PictureBot.sln that includes additions for Regex, LUIS and Azure Search. If you fall behind or get stuck, you can refer to this.

You need Visual Studio to run these labs, but if you have already deployed a Windows Data Science Virtual Machine for one of the workshops, you could use that.

Collecting the Keys

Over the course of this lab, we will collect various keys. It is recommended that you save all of them in a text file, so you can easily access them throughout the workshop.

> *Keys* - LUIS API: - Cosmos DB Connection String: - Azure Search Name: - Azure Search Key: - Bot Framework App Name: - Bot Framework App ID: - Bot Framework App Password:

Navigating the Labs

This workshop has been broken down into five sections: - 1_Regex_and_ScorableGroups: Here you will build a simple bot using Regular Expressions and Scorable Groups - 2_Azure_Search: We'll configure our bot for Azure Search and connect it to the Azure Search service from the previous lab - 3_LUIS: Next, we'll incorporate our LUIS model into our bot, so that we can call LUIS when Regex does not recognize a user's intent. - 4_Publish_and_Register: We'll finish by publishing and registering our bot. - 5_Challenge_and_Closing: If you get through all the labs, try this challenge. You will also find a summary of what you've done and where to learn more.

Continue to 1_Regex_and_ScorableGroups

1_Regex_and_ScorableGroups:

Estimated Time: 10-15 minutes

Building a Bot

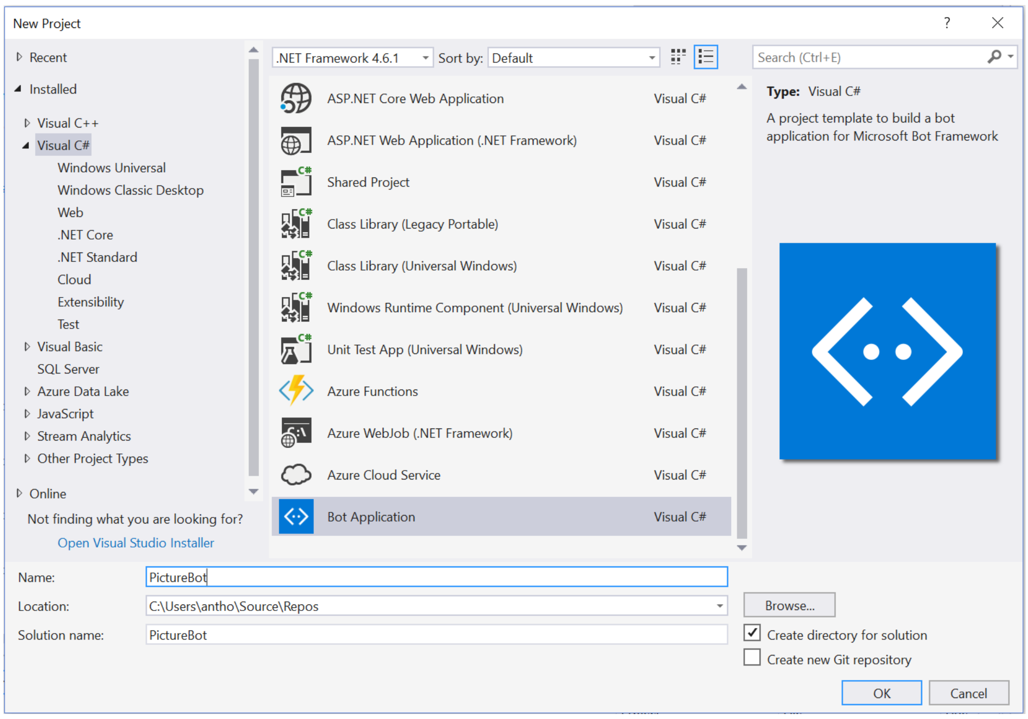
We assume that you've had some exposure to the Bot Framework. If you have, great. If not, don't worry too much, you'll learn a lot in this section. We recommend completing [this Microsoft Virtual Academy course]() and checking out the [documentation]().

Lab 1.1: Setting up for bot development

We will be developing a bot using the C# SDK. To get started, you need two things: 1. The Bot Framework project template, which you can download [here](). The file is called "Bot Application.zip" and you should save it into the \Documents\Visual Studio 2019\Templates\ProjectTemplates\Visual C#\ directory. Just drop the whole zipped file in there; no need to unzip.
2. Download the Bot Framework Emulator for testing your bot locally [here](). The emulator installs to `c:\Users\`*your-username*`\AppData\Local\botframework\app-3.5.33\botframework-emulator.exe` or your Downloads folder, depending on browser.

Lab 1.2: Creating a simple bot and running it

In Visual Studio, go to File --> New Project and create a Bot Application named "PictureBot". Make sure you name it "PictureBot" or you may have issues later on.


New Project

Recent
Installed
Visual C++
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Extensibility
Test
Visual Basic
SQL Server
Azure Data Lake
JavaScript
Stream Analytics
Other Project Types
Online
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.1
Sort by: Default
Search (Ctrl+E)
ASP.NET Core Web Application Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
Class Library (Universal Windows) Visual C#
Windows Runtime Component (Universal Windows) Visual C#
Unit Test App (Universal Windows) Visual C#
Azure Functions Visual C#
Azure WebJob (.NET Framework) Visual C#
Azure Cloud Service Visual C#
Bot Application Visual C#
Browse...
OK
Cancel

Type: Visual C#

A project template to build a bot application for Microsoft Bot Framework

Name: PictureBot

Location: C:\Users\antho\Source\Repos

Solution name: PictureBot

[✓] Create directory for solution

[ ] Create new Git repository

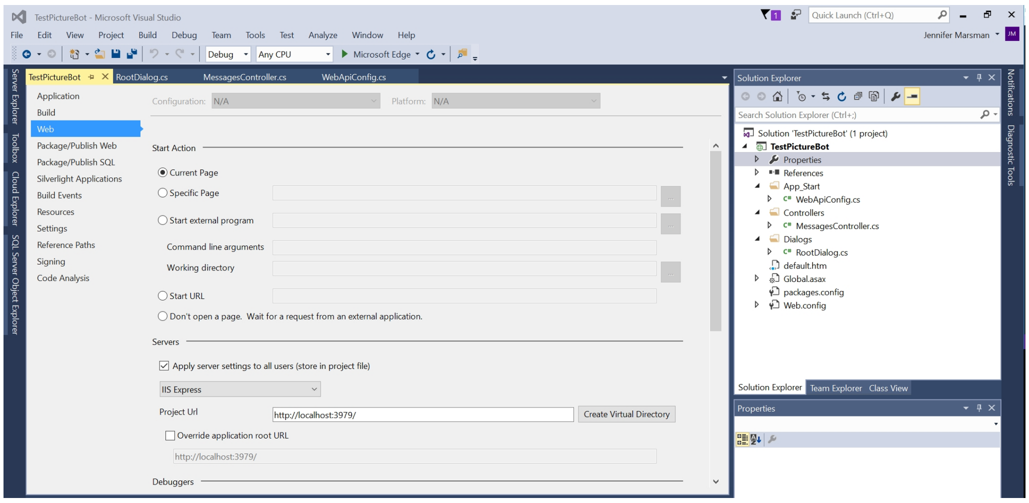
The rest of the **Creating a simple bot and running it** lab is optional. Per the prerequisites, you should have experience working with the Bot Framework. You can hit F5 to confirm it builds correctly, and move on to the next lab.

Browse around and examine the sample bot code, which is an echo bot that repeats back your message and its length in characters. In particular, **Note** + In **WebApiConfig.cs** under App_Start, the route template is api/{controller}/{id} where the id is optional. That is why we always call the bot's endpoint with api/messages appended at the end.
+ The **MessagesController.cs** under Controllers is therefore the entry point into your bot. Notice that a bot can respond to many different activity types, and sending a message will invoke the RootDialog.
+ In **RootDialog.cs** under Dialogs, "StartAsync" is the entry point which waits for a message from the user, and "MessageReceivedAsync" is the method that will handle the message once received and then wait for further messages. We can use "context.PostAsync" to send a message from the bot back to the user.

Click F5 to run the sample code. NuGet should take care of downloading the appropriate dependencies.

The code will launch in your default web browser in a URL similar to http://localhost:3979/.

> Fun Aside: why this port number? It is set in your project properties. In your Solution Explorer, double-click "Properties" and select the "Web" tab. The Project URL is set in the "Servers" section.


TestPictureBot - Microsoft Visual Studio

Web

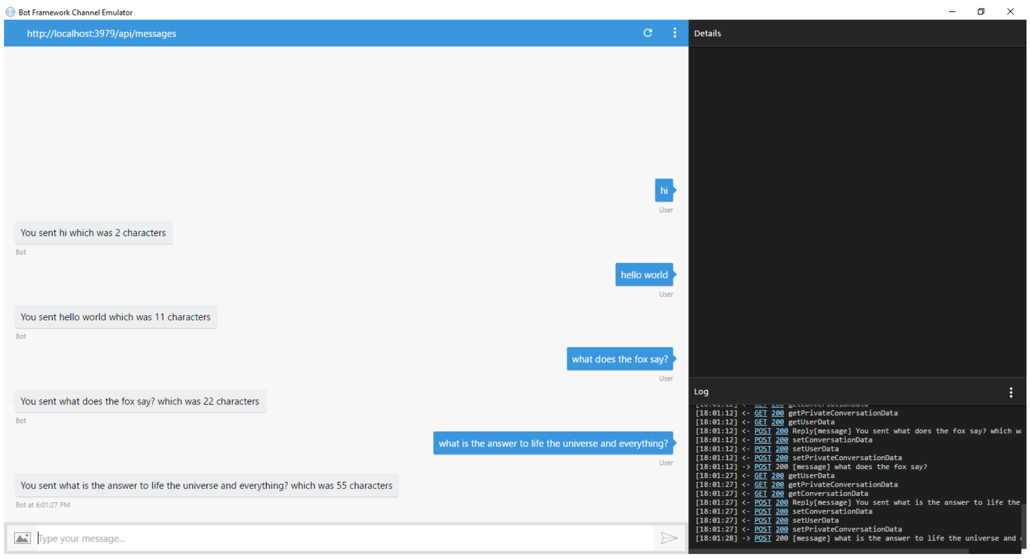
Configuration: N/A Platform: N/A

Start Action
- [✓ Current Page]
- Specific Page
- Start external program
- Command line arguments
- Working directory
- Start URL
- Don't open a page. Wait for a request from an external application.

Servers
- [✓] Apply server settings to all users (store in project file)
- IIS Express
- Project Url: http://localhost:3979/ Create Virtual Directory
- [ ] Override application root URL: http://localhost:3979/

Debuggers

Make sure your project is still running (hit F5 again if you stopped to look at the project properties) and launch the Bot Framework Emulator. (If you just installed it, it may not be indexed to show up in a search on your local machine, so remember that it installs to c:\Users\your-username\AppData\Local\botframework\app-3.5.27\botframework-emulator.exe.) Ensure that the Bot URL matches the port number that your code launched in above, and has api/messages appended to the end. You should be able to converse with the bot.


Bot Framework Channel Emulator
http://localhost:3979/api/messages
Details
Log
Type your message...

hi
User

You sent hi which was 2 characters
Bot

hello world
User

You sent hello world which was 11 characters
Bot

what does the fox say?
User

You sent what does the fox say? which was 22 characters
Bot

what is the answer to life the universe and everything?
User

You sent what is the answer to life the universe and everything? which was 55 characters
Bot at 6:01:27 PM

```
[18:01:12] <- GET 200 getPrivateConversationData
[18:01:12] <- GET 200 getUserData
[18:01:12] <- POST 200 Reply[message] You sent what does the fox say? which wa
[18:01:12] <- POST 200 setConversationData
[18:01:12] <- POST 200 setUserData
[18:01:12] <- POST 200 setPrivateConversationData
[18:01:12] -> POST 200 [message] what does the fox say?
[18:01:27] <- GET 200 getUserData
[18:01:27] <- GET 200 getPrivateConversationData
[18:01:27] <- GET 200 getConversationData
[18:01:27] <- POST 200 Reply[message] You sent what is the answer to life the
[18:01:27] <- POST 200 setConversationData
[18:01:27] <- POST 200 setUserData
[18:01:27] <- POST 200 setPrivateConversationData
[18:01:28] -> POST 200 [message] what is the answer to life the universe and
```

Lab 1.3: Regular expressions and scorable groups

There are a number of things that we can do to improve our bot. First of all, we may not want to call LUIS for a simple "hi" greeting, which the bot will get fairly frequently from its users. A simple regular expression could match this, and save us time (due to network latency) and money (due to cost of calling the LUIS service).

Also, as the complexity of our bot grows, and we are taking the user's input and using multiple services to interpret it, we need a process to manage that flow. For example, try regular expressions first, and if that doesn't match, call LUIS, and then perhaps we also drop down to try other services like QnA Maker and Azure Cognitive Search. A great way to manage this is ScorableGroups. ScorableGroups give you an attribute to impose an order on these service calls. In our code, let's impose an order of matching on regular expressions first, then calling LUIS for interpretation of utterances, and finally lowest priority is to drop down to a generic "I'm not sure what you mean" response.

To use ScorableGroups, your RootDialog will need to inherit from DispatchDialog instead of LuisDialog (but you can still have the LuisModel attribute on the class). You also will need a reference to Microsoft.Bot.Builder.Scorables (as well as others). So in your RootDialog.cs file, add:

```csharp

using Microsoft.Bot.Builder.Scorables; using System.Collections.Generic;

```

and change your class derivation to:

```csharp

```
public class RootDialog : DispatchDialog<object>
```
```

\`\`\`

Next, delete the two existing methods in the class (StartAsync and MessageReceivedAsync).

Let's add some new methods that match regular expressions as our first priority in ScorableGroup 0. Add the following at the beginning of your RootDialog class:

```csharp
    [RegexPattern("^Hello|hello")]
    [RegexPattern("^Hi|hi")]
    [ScorableGroup(0)]
    public async Task Hello(IDialogContext context, IActivity activity)
    {
        await context.PostAsync("Hello from RegEx!  I am a Photo
Organization Bot.  I can search your photos, share your photos on
Twitter, and order prints of your photos.  You can ask me things like
'find pictures of food'.");
    }

    [RegexPattern("^Help|help")]
    [ScorableGroup(0)]
    public async Task Help(IDialogContext context, IActivity activity)
    {
        // Launch help dialog with button menu
        List<string> choices = new List<string>(new string[] { "Search
Pictures", "Share Picture", "Order Prints" });
        PromptDialog.Choice<string>(context, ResumeAfterChoice,
            new PromptOptions<string>("How can I help you?",
options:choices));
    }

    private async Task ResumeAfterChoice(IDialogContext context,
IAwaitable<string> result)
    {
        string choice = await result;

        switch (choice)
        {
            case "Search Pictures":
                // PromptDialog.Text(context,
ResumeAfterSearchTopicClarification,
                //     "What kind of picture do you want to search
for?");
                break;
            case "Share Picture":
                //await SharePic(context, null);
                break;
            case "Order Prints":
                //await OrderPic(context, null);
                break;
            default:
                await context.PostAsync("I'm sorry. I didn't understand
you.");
                break;
        }
    }
```

```

This code will match on expressions from the user that start with "hi", "hello", and "help". Select F5 and test your bot. Notice that when the user asks for help, we present him/her with a simple menu of buttons on the three core things our bot can do: search pictures, share pictures, and order prints.

> Fun Aside: One might argue that the user shouldn't have to type "help" to get a menu of clear options on what the bot can do; rather, this should be the default experience on first contact with the bot. **Discoverability** is one of the biggest challenges for bots - letting the users know what the bot is capable of doing. Good bot design principles can help.

This setup will make it easier when we resort to LUIS (later) as our second attempt if no regular expression matches, in Scorable Group 1.

## Continue to 2_Azure_Search

Back to [README
```

2 Azure Cognitive Search

Estimated Time: 10-15 minutes

Lab 2.1: Configure your bot for Azure Cognitive Search

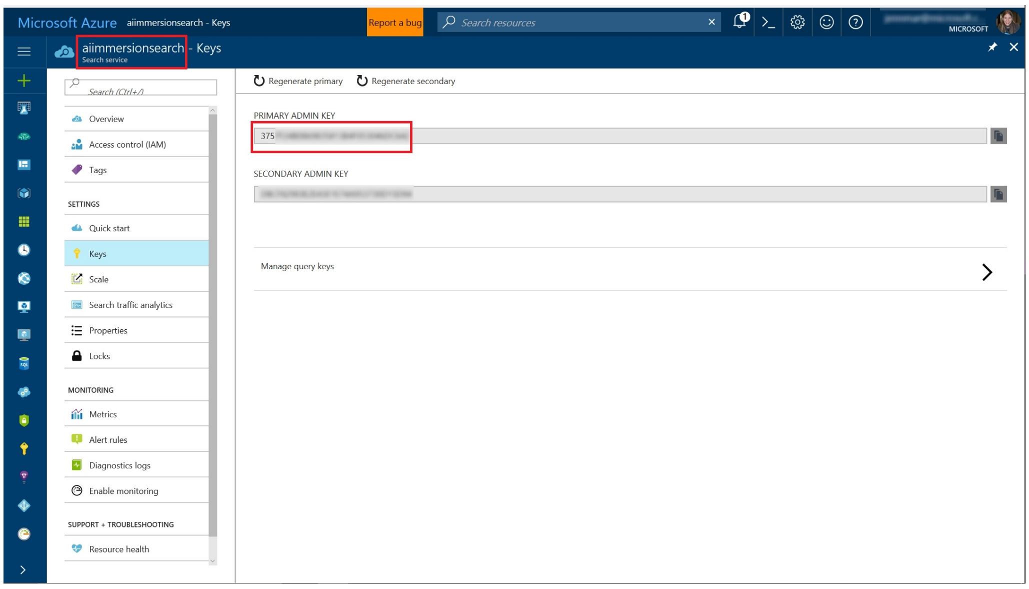
First, we need to provide our bot with the relevant information to connect to an Azure Cognitive Search index. The best place to store connection information is in the configuration file.

Open Web.config and in the appSettings section, add the following:

```
xml <!-- Azure Cognitive Search Settings --> <add
key="SearchDialogsServiceName" value="" /> <add
key="SearchDialogsServiceKey" value="" /> <add
key="SearchDialogsIndexName" value="images" />
```

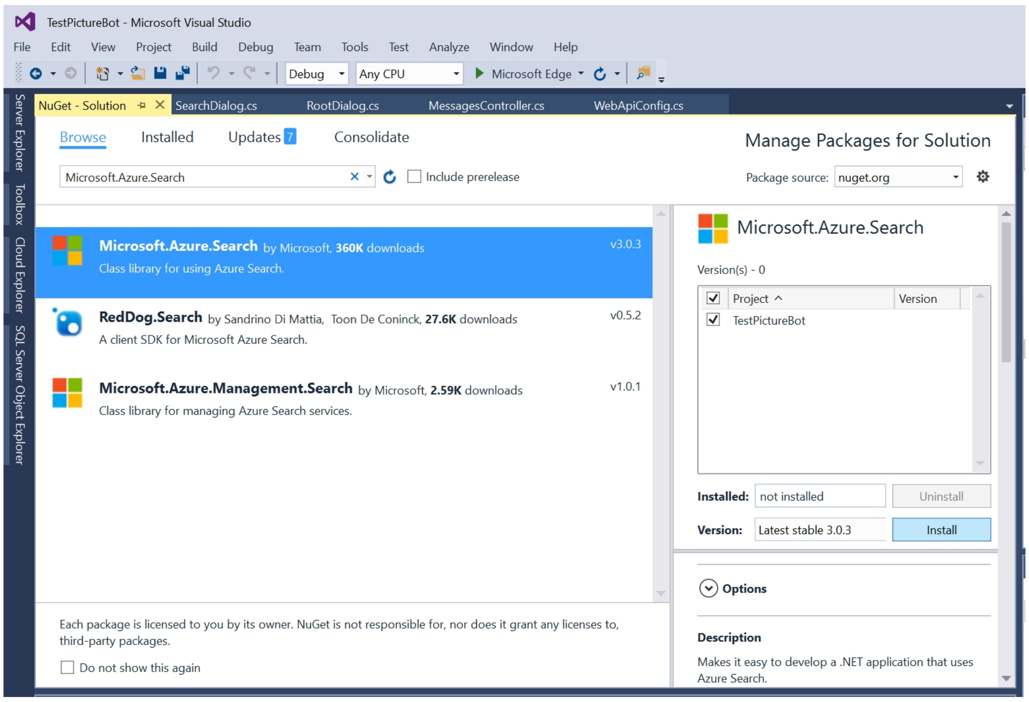
Set the value for the SearchDialogsServiceName to be the name of the Azure Cognitive Search Service that you created earlier. If needed, go back and look this up in the [Azure portal](https://portal.azure.com).

Set the value for the SearchDialogsServiceKey to be the key for this service. This can be found in the [Azure portal](https://portal.azure.com) under the Keys section for your Azure Cognitive Search. In the below screenshot, the SearchDialogsServiceName would be "aiimmersionsearch" and the SearchDialogsServiceKey would be "375...".



Lab 2.2: Update the bot to use Azure Cognitive Search

Next, we'll update the bot to call Azure Cognitive Search. First, open Tools-->NuGet Package Manager-->Manage NuGet Packages for Solution. In the search box, type "Microsoft.Azure.Search". Select the corresponding library, check the box that indicates your project, and install it. It may install other dependencies as well. Under installed packages, you may also need to update the "Newtonsoft.Json" package.



Microsoft.Azure.Search

Microsoft.Azure.Search by Microsoft, **360K** downloads v3.0.3
Class library for using Azure Search.

RedDog.Search by Sandrino Di Mattia, Toon De Coninck, **27.6K** downloads v0.5.2
A client SDK for Microsoft Azure Search.

Microsoft.Azure.Management.Search by Microsoft, **2.59K** downloads v1.0.1
Class library for managing Azure Search services.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

☐ Do not show this again

Microsoft.Azure.Search

Version(s) - 0

☑ TestPictureBot

Installed: not installed

Version: Latest stable 3.0.3

Description

Makes it easy to develop a .NET application that uses Azure Search.

Right-click on your project in the Solution Explorer of Visual Studio, and select Add-->New Folder. Create a folder called "Models". Then right-click on the Models folder, and select Add-->Existing Item. Do this twice to add these two files under the Models folder (make sure to adjust your namespaces if necessary): 1. ImageMapper.cs 2. SearchHit.cs

You can find the files in this repository under resources/code/Models

Next, right-click on the Dialogs folder in the Solution Explorer of Visual Studio, and select Add-->Class. Call your class "SearchDialog.cs". Add the contents from here.

Review the contents of the files you just added. Discuss with a neighbor what each of them does.

We also need to update your RootDialog to call the SearchDialog. In RootDialog.cs in the Dialogs folder, directly under the `ResumeAfterChoice` method, add these "ResumeAfter" methods:

```csharp
    private async Task
ResumeAfterSearchTopicClarification(IDialogContext context,
IAwaitable<string> result)
    {
        string searchTerm = await result;
        context.Call(new SearchDialog(searchTerm),
ResumeAfterSearchDialog);
    }
    private async Task ResumeAfterSearchDialog(IDialogContext context,
IAwaitable<object> result)
    {
        await context.PostAsync("Done searching pictures");
    }
```

```

In RootDialog.cs, you will also need to remove the comments (the `//` at the beginning) from the line: `PromptDialog.Text(context, ResumeAfterSearchTopicClarification, "What kind of picture do you want to search for?");` within the `ResumeAfterChoice` method.

Press F5 to run your bot again. In the Bot Emulator, try searching for something like "dogs" or "water". Ensure that you are seeing results when tags from your pictures are requested.

## Continue to 3_LUIS

Back to [README
```

3_LUIS:

Estimated Time: 15-20 minutes

Lab 3.1: Update bot to use LUIS

We have to update our bot in order to use LUIS. We can do this by using the LuisDialog class.

In the **RootDialog.cs** file, add references to the following namespaces:

```csharp

using Microsoft.Bot.Builder.Luis; using Microsoft.Bot.Builder.Luis.Models;

```

Next, give the class a LuisModel attribute with the LUIS App ID and LUIS key. If you can't find these values, go back to http://luis.ai. Click on your application, and go to the "Publish App" page. You can get the LUIS App ID and LUIS key from the Endpoint URL (HINT: The LUIS App ID will have hyphens in it, and the LUIS key will not). You will need to put the following line of code directly above the `[Serializable]` with your LUIS App ID and LUIS key:

```csharp

namespace PictureBot.Dialogs { [LuisModel("YOUR-APP-ID", "YOUR-SUBSCRIPTION-KEY")] [Serializable]

... ```

> Fun Aside: You can use Autofac to dynamically load the LuisModel attribute on your class instead of hardcoding it, so it could be stored properly in a configuration file. There is an example of this in the AlarmBot sample.

Let's start simple by adding our "Greeting" intent (below all of the code you've already put within the DispatchDialog):

`csharp [LuisIntent("Greeting")] [ScorableGroup(1)] public async Task Greeting(IDialogContext context, LuisResult result) { // Duplicate logic, for a teachable moment on Scorables. await context.PostAsync("Hello from LUIS! I am a Photo Organization Bot. I can search your photos, share your photos on Twitter, and order prints of your photos. You can ask me things like 'find pictures of food'."); }`

Hit F5 to run the app. In the Bot Emulator, try sending the bots different ways of sending hello. What happens when you send "whats up" versus "hello"?

The "None" intent in LUIS means that the utterance didn't map to any intent. In this situation, we want to fall down to the next level of ScorableGroup. Add the "None"
```

method in the RootDialog class as follows: `csharp [LuisIntent("")] [LuisIntent("None")] public async Task None(IDialogContext context, LuisResult result) { // Luis returned with "None" as the winning intent, // so drop down to next level of ScorableGroups. ContinueWithNextGroup(); }`

Finally, add methods for the rest of the intents. The corresponding method will be invoked for the highest-scoring intent. Talk with a neighbor about the "SearchPics" and "SharePics" code. What else is the code doing?

```csharp [LuisIntent("SearchPics")] public async Task SearchPics(IDialogContext context, LuisResult result) { // Check if LUIS has identified the search term that we should look for.
string facet = null; EntityRecommendation rec; if (result.TryFindEntity("facet", out rec)) facet = rec.Entity;

```
        // If we don't know what to search for (for example, the user
said
        // "find pictures" or "search" instead of "find pictures of x"),
        // then prompt for a search term.
        if (string.IsNullOrEmpty(facet))
        {
            PromptDialog.Text(context,
ResumeAfterSearchTopicClarification,
                "What kind of picture do you want to search for?");
        }
        else
        {
            await context.PostAsync("Searching pictures...");
            context.Call(new SearchDialog(facet),
ResumeAfterSearchDialog);
        }
    }

    [LuisIntent("OrderPic")]
    public async Task OrderPic(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Ordering your pictures...");
    }

    [LuisIntent("SharePic")]
    public async Task SharePic(IDialogContext context, LuisResult
result)
    {
        PromptDialog.Confirm(context, AfterShareAsync,
            "Are you sure you want to tweet this picture?");
    }

    private async Task AfterShareAsync(IDialogContext context,
IAwaitable<bool> result)
    {
        if (result.GetAwaiter().GetResult() == true)
        {
            // Yes, share the picture.
            await context.PostAsync("Posting tweet.");
        }
        else
```
```

```
        {
            // No, don't share the picture.
            await context.PostAsync("OK, I won't share it.");
        }
    }
```

> Hint: The "SharePic" method contains a little code to show how to do a prompt for a yes/no confirmation as well as setting the ScorableGroup. This code doesn't actually post a tweet because we didn't want to spend time getting everyone set up with Twitter developer accounts and such, but you are welcome to implement if you want.

You may have noticed that we haven't implemented scorable groups for the intents we just added. Modify your code so that all of the `LuisIntent`s are of scorable group priority 1. Now that you've added LUIS functionality, you can uncomment the two `await` lines in the `ResumeAfterChoice` method.

Once you've modified your code, hit F5 to run in Visual Studio, and start up a new conversation in the Bot Framework Emulator. Try chatting with the bot, and ensure that you get the expected responses. If you get any unexpected results, note them down and revise LUIS.

Don't forget, you must re-train and re-publish your LUIS model. Then you can return to your bot in the emulator and try again.

> Fun Aside: The Suggested Utterances are extremely powerful. LUIS makes smart decisions about which utterances to surface. It chooses the ones that will help it improve the most to have manually labeled by a human-in-the-loop. For example, if the LUIS model predicted that a given utterance mapped to Intent1 with 47% confidence and predicted that it mapped to Intent2 with 48% confidence, that is a strong candidate to surface to a human to manually map, since the model is very close between two intents.

> Extra credit (to complete later): create an OrderDialog class in your "Dialogs" folder. Create a process for ordering prints with the bot using FormFlow. Your bot will need to collect the following information: Photo size (8x10, 5x7, wallet, etc.), number of prints, glossy or matte finish, user's phone number, and user's email.

Finally, add a default handler if none of the above services were able to understand. This ScorableGroup needs an explicit MethodBind because it is not decorated with a LuisIntent or RegexPattern attribute (which include a MethodBind).

```csharp
    // Since none of the scorables in previous group won, the dialog
sends a help message.
    [MethodBind]
    [ScorableGroup(2)]
    public async Task Default(IDialogContext context, IActivity
activity)
    {
        await context.PostAsync("I'm sorry. I didn't understand you.");
        await context.PostAsync("You can tell me to find photos, tweet
them, and order prints.  Here is an example: \"find pictures of
food\".");
    }

```

Hit F5 to run your bot and test it in the Bot Emulator.

Continue to 4_Publish_and_Register

Back to [README

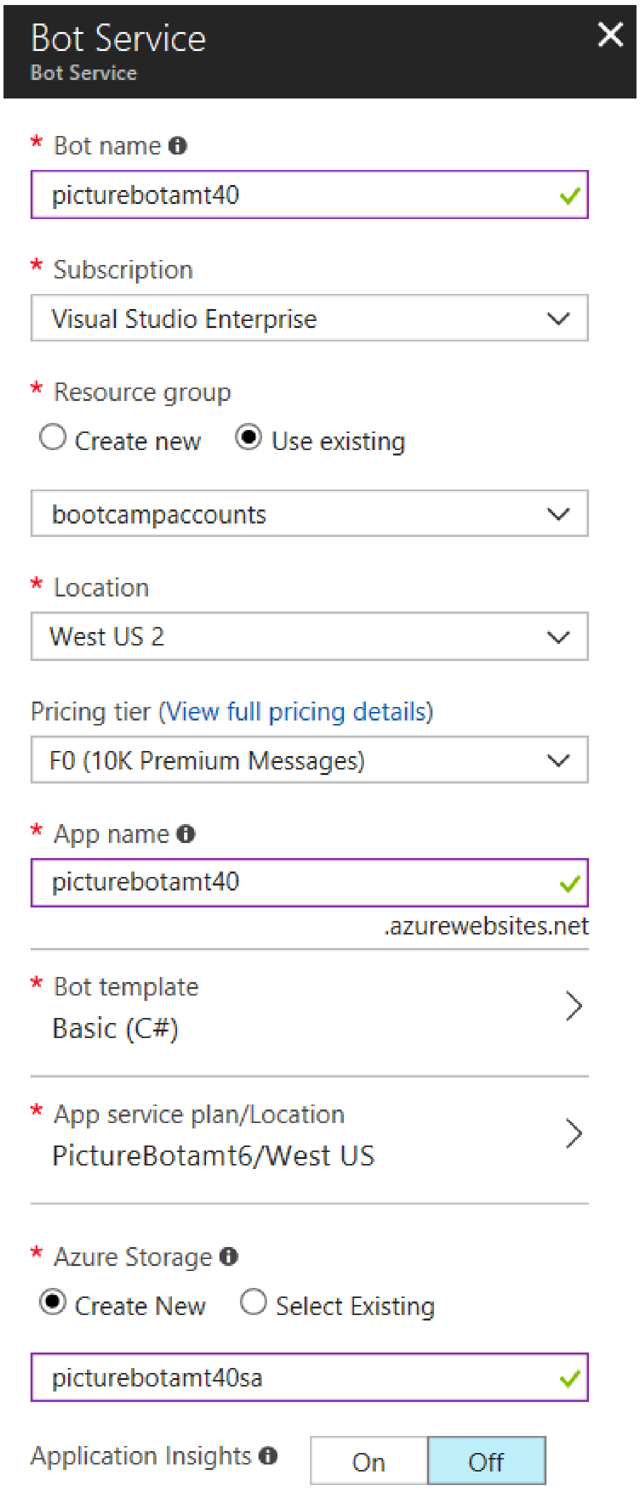
4_Publish_and_Register:

Estimated Time: 10-15 minutes

Lab 4.1: Publish your 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 PictureBotamt40). Put in the region that is closest to you. For pricing tier, select F0, as that is all we will need for this workshop. Set the bot template to Basic (C#), and configure a new App service plan (put it in the same location as your bot). It doesn't matter which template you choose, because we will overwrite it with our PictureBot. Turn off Application Insights (to save money). Click create.


Bot Service
Bot Service

* Bot name
picturebotamt40 ✓

* Subscription
Visual Studio Enterprise

* Resource group
Create new [✓ Use existing]

bootcampaccounts

* Location
West US 2

Pricing tier (View full pricing details)
F0 (10K Premium Messages)

* App name
picturebotamt40 ✓
.azurewebsites.net

* Bot template
Basic (C#)

* App service plan/Location
PictureBotamt6/West US

* Azure Storage
[✓ Create New] Select Existing

picturebotamt40sa ✓

Application Insights On [✓ Off]

You have just published a very simple EchoBot (like the template we started from earlier). What we will do next is publish our PictureBot to this bot service.

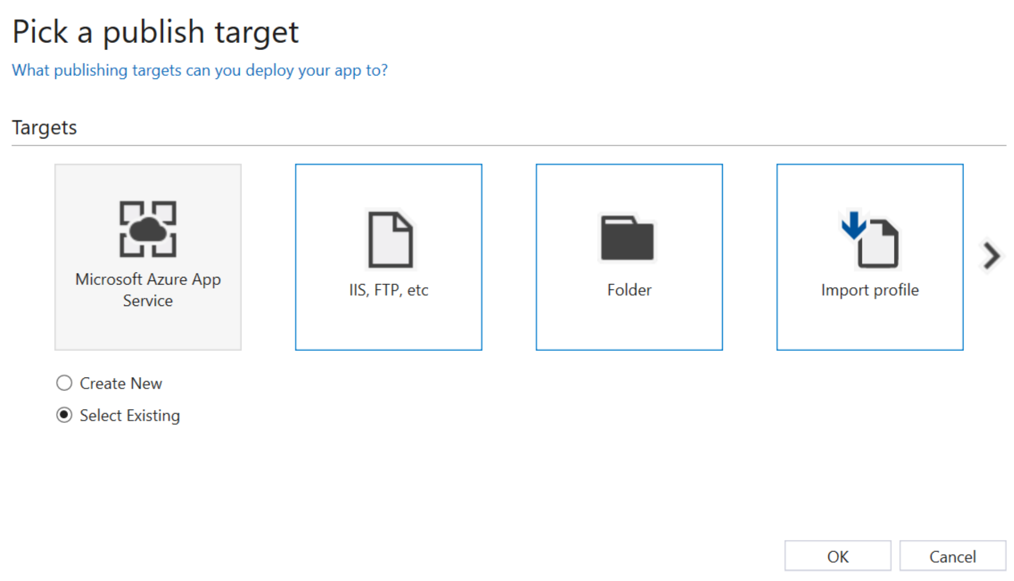
First we need to grab a few keys. Go to the Web App Bot you just created (in the portal). Under App Service Settings->Application Settings->App settings, grab the BotId, MicrosoftAppId, and MicrosoftAppPassword. You will need them in a moment.

Return to your PictureBot in Visual Studio. In the Web.config file, fill in the the blanks under `appSettings` with the BotId, MicrosoftAppId, and MicrosoftAppPassword. Save the file.

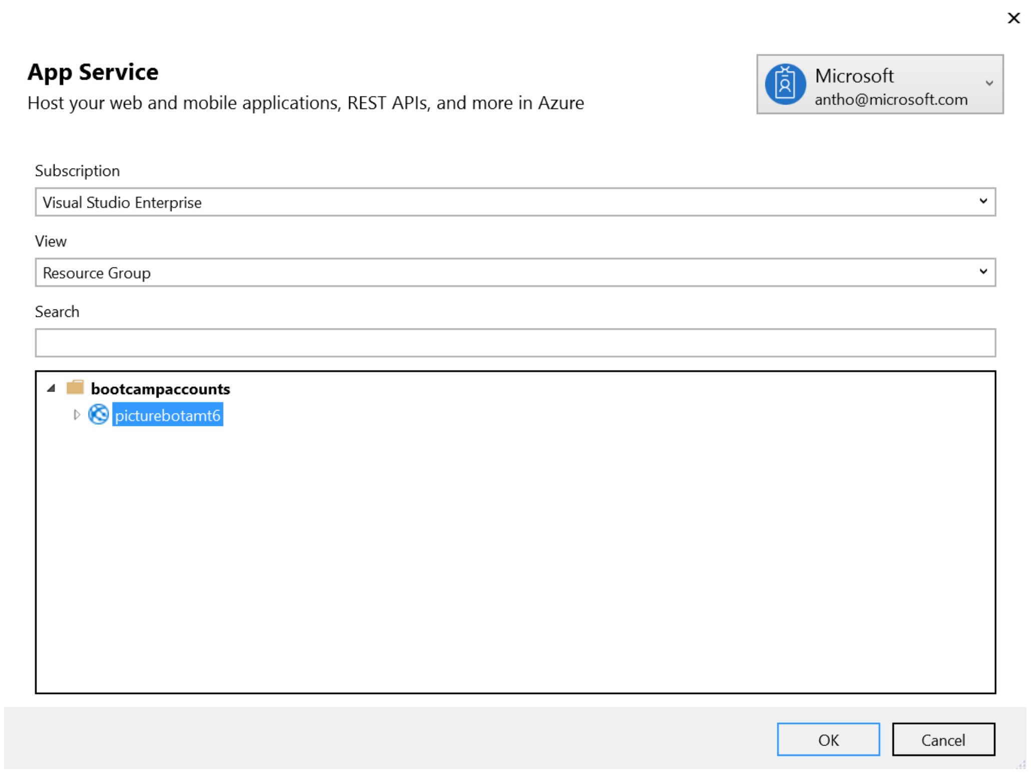
> Getting an error that directs you to your MicrosoftAppPassword? Because it's in XML, if your key contains "&", "<", ">", """, or "'", you will need to replace those symbols with their respective escape facilities: "\&", "\<", "\>", "\'", "\"".

In the Solution Explorer, right-click on your Bot Application project and select "Publish". This will launch a wizard to help you publish your bot to Azure.

Select the publish target of "Microsoft Azure App Service" and "Select Existing."



On the App Service screen, select the appropriate subscription, resource group, and your Bot Service. Select Publish.


App Service

Host your web and mobile applications, REST APIs, and more in Azure

Microsoft
antho@microsoft.com

Subscription
Visual Studio Enterprise

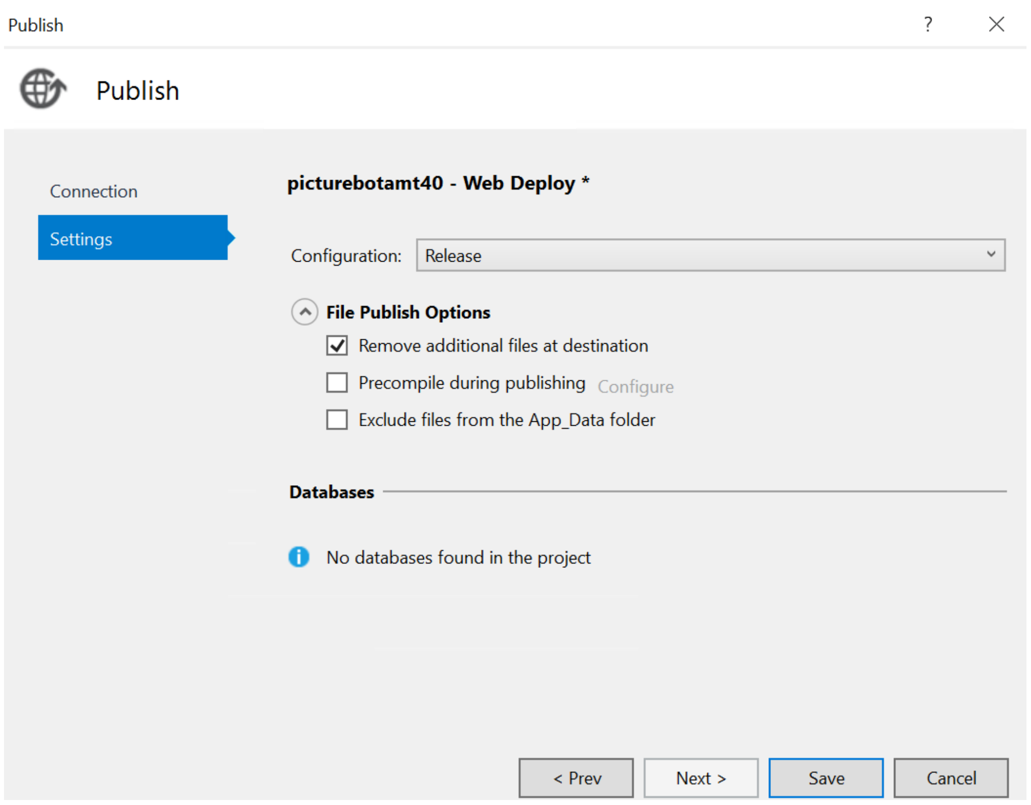
View
Resource Group

Search

bootcampaccounts
picturebotamt6

If you get an error here, simply exit the browser window within Visual Studio and complete the next step.

Now, you will see the Web Deploy settings, but we have to edit them one last time. Select "Settings" under the publish profile. Select Settings again and check the box next to "Remove additional files at destination". Click "Save" to exit the window, and now click "Publish". The output window in Visual Studio will show the deployment process. Then, your bot will be hosted at a URL like http://picturebotamt6.azurewebsites.net/, where "picturebotamt6" is the Bot Service app name.


Publish

Publish

Connection

Settings

picturebotamt40 - Web Deploy *

Configuration: Release

File Publish Options

[✓] Remove additional files at destination

[ ] Precompile during publishing Configure

[ ] Exclude files from the App_Data folder

Databases

No databases found in the project

Managing your bot from the portal

Return to the portal to your Web App Bot resource. Under Bot Management, select "Test in Web Chat" to test if your bot has been published and is working accordingly. If it is not, review the previous lab, because you may have skipped a step. Re-publish and return here.

After you've confirmed your bot is published and working, check out some of the other features under Bot Management. Select "Channels" and notice there are many channels, and when you select one, you are instructed on how to configure it.

Want to learn more about the various aspects related to bots? Spend some time reading the how to's and design principles.

Continue to 5_Challenge_and_Closing

Back to [README

5_Challenge_and_Closing

Finish early? Try this extra credit lab:

Try experimenting with more advanced Azure Cognitive Search queries. Add term-boosting by extending your LUIS model to recognize entities like *"find happy people"*, mapping "happy" to "happiness" (the emotion returned from Cognitive Services), and turning those into boosted queries using Term Boosting.

Lab Completion

In this lab we covered creating an intelligent bot from end-to-end using the Microsoft Bot Framework, Azure Cognitive Search and several Cognitive Services.

You should have learned: - How to weave intelligent services into your applications - How to implement Azure Cognitive Search features to provide a positive search experience inside an application - How to configure an Azure Cognitive Search service to extend your data to enable full-text, language-aware search - How to build, train and publish a LUIS model to help your bot communicate effectively - How to build an intelligent bot using Microsoft Bot Framework that leverages LUIS and Azure Cognitive Search

Resources for future projects/learning: - Azure Bot Services documentation - Azure Cognitive Search documentation - Azure Bot Builder Samples - Azure Cognitive Search Samples - LUIS documentation - LUIS Sample - Course on Designing and Architecting Intelligent Agents

Back to [README

lab02.3-luis_and_search - Developing Intelligent Applications with LUIS and Azure Cognitive Search

This hands-on lab guides you through creating an intelligent bot from end-to-end using the Microsoft Bot Framework, Azure Cognitive Search and several Cognitive Services.

In this workshop, you will: - Understand how to weave intelligent services into your applications - Understand how to implement Azure Cognitive Search features to provide a positive search experience inside applications - Configure an Azure Cognitive Search service to extend your data to enable full-text, language-aware search - Build, train and publish a LUIS model to help your bot communicate effectively - Build an intelligent bot using Microsoft Bot Framework that leverages LUIS and Azure Cognitive Search - Call various Cognitive Services APIs (specifically Computer Vision, Face, Emotion and LUIS) in .NET applications

While there is a focus on LUIS and Azure Cognitive Search, you will also leverage the following technologies:

- Computer Vision API
- Face API
- Emotion API
- Data Science Virtual Machine (DSVM)
- Windows 10 SDK (UWP)
- CosmosDB
- Azure Storage
- Visual Studio

Prerequisites

This workshop is meant for an AI Developer on Azure. Since this is a half-day workshop, there are certain things you need before you arrive.

Firstly, you should have experience with Visual Studio. We will be using it for everything we are building in the workshop, so you should be familiar with how to use it to create applications. Additionally, this is not a class where we teach you how to code or develop applications. We assume you know how to code in C# (you can learn here), but you do not know how to implement advanced Search and NLP (natural language processing) solutions.

Secondly, you should have some experience developing bots with Microsoft's Bot Framework. We won't spend a lot of time discussing how to design them or how dialogs work. If you are not familiar with the Bot Framework, you should take this Microsoft Virtual Academy course prior to attending the workshop.

Thirdly, you should have experience with the portal and be able to create resources (and spend money) on Azure. We will not be providing Azure passes for this workshop.

Intro

We're going to build an end-to-end scenario that allows you to pull in your own pictures, use Cognitive Services to find objects and people in the images, figure out how those people are feeling, and store all of that data into a NoSQL Store (DocumentDB). We use that NoSQL Store to populate an Azure Cognitive Search index, and then build a Bot Framework bot using LUIS to allow easy, targeted querying.

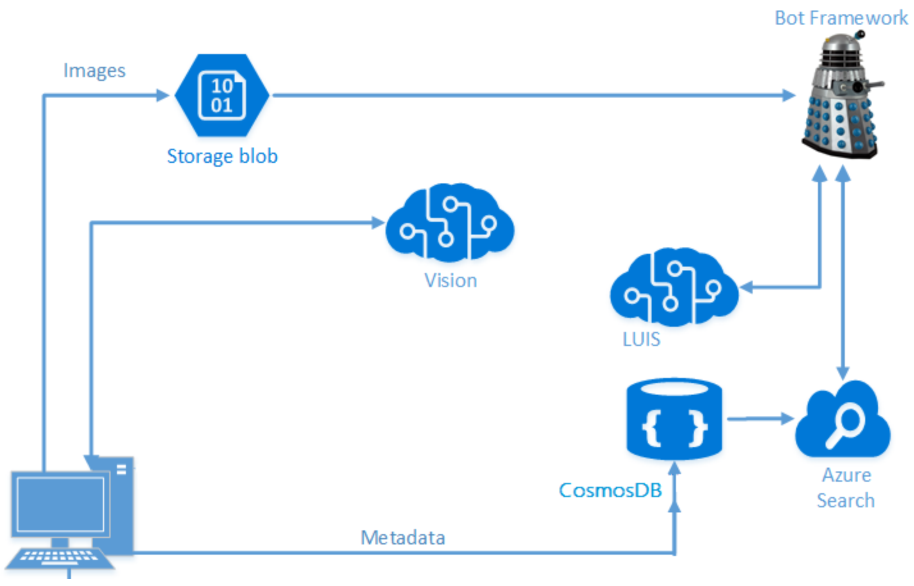
Architecture

We will build a simple C# application that allows you to ingest pictures from your local drive, then invoke several different Cognitive Services to gather data on those images:

- Computer Vision: We use this to grab tags and a description
- Face: We use this to grab faces and their details from each image
- Emotion: We use this to pull emotion scores from each face in the image

Once we have this data, we process it to pull out the details we need, and store it all into DocumentDB, our NoSQL PaaS offering.

Once we have it in DocumentDB, we'll build an Azure Cognitive Search Index on top of it (Azure Cognitive Search is our PaaS offering for faceted, fault-tolerant search - think Elastic Search without the management overhead). We'll show you how to query your data, and then build a Bot Framework bot to query it. Finally, we'll extend this bot with LUIS to automatically derive intent from your queries and use those to direct your searches intelligently.



Navigating the GitHub

There are several directories in the resources folder:

- **assets**: This contains all of the images for the lab manual. You can ignore this folder.

- **code**: In here, there are several directories that we will use:

  - **ImageProcessing**: This solution (.sln) contains several different projects for the first parts of the workshop, let's take a high level look at them:

    - **ImageProcessingLibrary**: This is a Portable Class Library (PCL) containing helper classes for accessing the various Cognitive Services related to Vision, and some "Insights" classes for encapsulating the results.
    - **ImageStorageLibrary**: Since Cosmos DB does not (yet) support UWP, this is a non-portable library for accessing Blob Storage and Cosmos DB.
    - **TestApp**: A UWP application that allows you to load your images and call the various cognitive services on them, then explore the results. Useful for experimentation and exploration of your images.
    - **TestCLI**: A Console application allowing you to call the various cognitive services and then upload the images and data to Azure. Images are uploaded to Blob Storage, and the various metadata (tags, captions, faces) are uploaded to Cosmos DB.

    Both *TestApp* and *TestCLI* contain a `settings.json` file containing the various keys and endpoints needed for accessing the Cognitive Services and Azure. They start blank, so once you provision your resources, we will grab your service keys and set up your storage account and Cosmos DB instance.

  - **LUIS**: Here you will find the LUIS model for the PictureBot. You will create your own, but if you fall behind or want to test out a different LUIS model, you can use the .json file to import this LUIS app.

  - **Models**: These classes will be used when we add search to our PictureBot.
  - **PictureBot**: Here there is a PictureBot.sln that is for the latter sections of the workshop, where we integrate LUIS and our Search Index into the Bot Framework

Lab: Setting 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.

Lab: Setting up your Data Science Virtual Machine

After creating an Azure account, you may access the Azure portal. From the portal, create a Resource Group for this lab. Detailed information about the Data Science Virtual Machine can be found online, but we will just go over what's needed for this workshop. In your Resource Group, deploy and connect to a Data Science Virtual Machine for Windows (2016), with a size of DS3_V2 (all other defaults are fine).

Once you're connected, there are several things you need to do to set up the DSVM for the workshop:

1. Navigate to this repository in Firefox, and download it as a zip file. Extract all the files, and move the folder for this lab to your Desktop.
2. Open "ImageProcessing.sln" which is under resources>code>ImageProcessing. It may take a while for Visual Studio to open for the first time, and you will have to log in.
3. Once it's open, you will be prompted to install the SDK for Windows 10 App Development (UWP). Follow the prompts to install it. If you aren't prompted, right-click on TestApp and select "Reload project", then you will be prompted.
4. While it's installing, there are a few tasks you can complete:
   - Type in the Cortana search bar "For developers settings" and change the settings to "Developer Mode".
   - Type in the Cortana search bar "gpedit.msc" and push enter. Enable the following policy: Computer Configuration>Windows Settings>Security Settings>Local Policies>Security Options>User Account Control: Admin Approval Mode for the Built-in Administrator account
   - Start the "Collecting the keys" lab.
5. Once the install is complete and you have changed your devleoper settings and the User Account Control policy, reboot your DSVM.

   > **Note** Be sure to turn off your DSVM after the workshop so you don't get charged.

Lab: Collecting 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: - Face API: - Emotion API: - LUIS API:

> *Storage Keys* - Azure Blob Storage Connection String: - Cosmos DB URI: - Cosmos DB key:

Getting Cognitive Services API Keys

Within the Portal, we'll first create keys for the Cognitive Services we'll be using. We'll primarily be using different APIs under the Computer Vision Cognitive Service, so let's create an API key for that first.

In the Portal, hit **New** and then enter **cognitive** in the search box and choose **Cognitive Services**:

Creating a Cognitive Service Key

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. We'll be using S1 so that we have the throughput we need for the tutorial, and creating a new *Resource Group*. We'll be using this same resource group below for our Blob Storage and Cosmos DB. *Pin to dashboard* so that you can easily find it. Since the Computer Vision API stores images internally at Microsoft (in a secure fashion), to help improve future Cognitive Services Vision offerings, you'll need to *Enable* Account creation. This can be a stumbling block for users in Enterprise environment, as only Subscription Administrators have the right to enable this, but for Azure Pass users it's not an issue.

Choosing Cognitive Services Details

Once you have created your new API subscription, you can grab the keys from the appropriate section of the blade, and add them to your *TestApp's* and *TestCLI's* `settings.json` file.

Cognitive API Key

We'll also be using other APIs within the Computer Vision family, so take this opportunity to create API keys for the *Emotion* and *Face* APIs as well. They are created in the same fashion as above, and should re-use the same Resource Group you've created. *Pin to Dashboard*, and then add those keys to your `settings.json` files.

Since we'll be using LUIS later in the tutorial, let's take this opportunity to create our LUIS subscription here as well. It's created in the exact same fashion as above, but choose Language Understanding Intelligent Service from the API drop-down, and re-use the same Resource Group you created above. Once again, *Pin to Dashboard* so once we get to that stage of the tutorial you'll find it easy to get access.

Setting 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 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 can be quickly integrated into Azure Cognitive Search.

Azure Blob Storage

Detailed "Getting Started" instructions can be found online, but let's just go over what you need for this lab.

Within the Azure Portal, click **New->Storage->Storage Account**

New Azure Storage

Once you click it, you'll be presented with the fields above to fill out. Choose your storage account name (lowercase letters and numbers), set *Account kind* to *Blob storage*, *Replication* to *Locally-Redundant storage (LRS)* (this is just to save money), use the same Resource Group as above, and set *Location* to *West US*. (The list of Azure services that are available in each region is at https://azure.microsoft.com/en-us/regions/services/). *Pin to dashboard* so that you can easily find it.

Now that you have an Azure Storage account, let's grab the *Connection String* and add it to your *TestCLI* and *TestApp* `settings.json`.

Azure Blob Keys

Cosmos DB

Detailed "Getting Started" instructions can be found online, but we'll walk through what you need for this project here.

Within the Azure Portal, click **New->Databases->Azure Cosmos DB**.

New Cosmos DB

Once you click this, you'll have to fill out a few fields as you see fit.

Cosmos DB Creation Form

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 Document DB SDK and not Mongo, so select Document DB as the NoSQL 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.

Once creation is complete, open the panel for your new database and select the *Keys* sub-panel.

Keys sub-panel for Cosmos DB

You'll need the **URI** and the **PRIMARY KEY** for your *TestCLI's* `settings.json` file, so copy those into there and you're now ready to store images and data into the cloud.

Cognitive Services

The focus of this workshop is not on all of the Cognitive Services APIs. We will be focusing on Azure Cognitive Search and LUIS. However, if you are interested in getting more practice with Cognitive Services, I recommend checking out -TODO: Add reference lab link-Lab 1, where you build an intelligent kiosk using several Cognitive Services.

For the purposes of this workshop, we will be using the Computer Vision API to get understanding of images (via tags and descriptions), the Face API to keep track of all of the faces in the images we upload, and the Emotion API to grab a score of which emotions people have in the images. We will simply call the API to get that information. There is no training or testing that we need to do.

The resulting information will include tags, a description, and gender/age/emotion if it's a person. After you've completed the following lab, check out the ImageInsights.json file that is created to see how this information is structured.

Later, we will also spend some time developing a LUIS model so that our bot has better language understanding. For this service, we will add intents, entities, utterances and more to our model before training and publishing it. Only then can we access it from an API call.

Lab: Exploring Cognitive Services and Image Processing Library

When you open the `ImageProcessing.sln` solution, you will find a UWP application (`TestApp`) that allows you to load your images and call the various Cognitive Services on them, then explore the results. It is useful for experimentation and exploration of your images. This app is built on top of the `ImageProcessingLibrary` project, which is also used by the TestCLI project to analyze the images.

You'll want to "Build" the solution (right click on `ImageProcessing.sln` and select "Build"). You also may have to reload the TestApp project, which you can do by right-clicking on it and selecting "Reload project".

Before running the app, make sure to enter the Cognitive Services API keys in the `settings.json` file under the `TestApp` project. Once you do that, run the app, point it to any folder (you will need to unzip `sample_images` first) with images (via the `Select Folder` button), and it should generate results like the following, showing all the images it processed, along with a break down of unique faces, emotions and tags that also act as filters on the image collection.

UWP Test App

Once the app processes a given directory it will cache the resuls in a `ImageInsights.json` file in that same folder, allowing you to look at that folder results again without having to call the various APIs.

Exploring Cosmos DB

Cosmos DB is not a focus of this workshop, but if you're interested in what's going on - here are some highlights from the code we will be using: - Navigate to the `DocumentDBHelper.cs` class in the `ImageStorageLibrary`. Many of the implementations we are using can be found in the Getting Started guide. - Go to `TestCLI`'s `Util.cs` and review the `ImageMetadata` class. This is where we turn the `ImageInsights` we retrieve from Cognitive Services into appropriate Metadata to be stored into Cosmos DB. - Finally, look in `Program.cs` and notice in `ProcessDirectoryAsync`. First, we check if the image and metadata have already been uploaded - we can use `DocumentDBHelper` to find the document by ID and to return `null` if the document doesn't exist. Next, if we've set `forceUpdate` or the image hasn't been processed before, we'll call the Cognitive Services using `ImageProcessor` from the `ImageProcessingLibrary` and retrieve the `ImageInsights`, which we add to our current `ImageMetadata`. - Once all of that is complete, we can store our image - first the actual image into Blob Storage using our `BlobStorageHelper` instance, and then the `ImageMetadata` into Cosmos DB using our `DocumentDBHelper` instance. If the document already existed (based on our previous check), we should update the existing document. Otherwise, we should be creating a new one.

Lab: 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 your ImageProcessing\TestCLI folder (Hint: use the "cd" command to change directories). Then enter `.\bin\Debug\TestCLI.exe`. You should get the following result:

```
> .\bin\Debug\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:

```
> .\bin\Debug\TestCLI.exe -process c:\my\image\directory
```

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

```
> .\bin\Debug\TestCLI.exe -query "select * from images"
```

Azure Cognitive Search

Azure Cognitive Search is a search-as-a-service solution allowing developers to incorporate great search experiences into applications without managing infrastructure or needing to become search experts.

Developers look for PaaS services in Azure to achieve better results faster in their apps. Search is a key to many categories of applications. Web search engines have set the bar high for search; users expect instant results, auto-complete as they type, highlighting hits within the results, great ranking, and the ability to understand what they are looking for, even if they spell it incorrectly or include extra words.

Search is a hard and rarely a core expertise area. From an infrastructure standpoint, it needs to have high availability, durability, scale, and operations. From a functionality standpoint, it needs to have ranking, language support, and geospatial capabilities.

Example of Search Requirements

The example above illustrates some of the components users are expecting in their search experience. Azure Cognitive Search can accomplish these user experience features, along with giving you monitoring and reporting, simple scoring, and tools for prototyping and inspection.

Typical Workflow: 1. Provision service - You can create or provision an Azure Cognitive Search service from the portal or with PowerShell. 2. 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. 3. Index 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 Cognitive Search REST API or .NET SDK. The second option is to point a supported data source to your index and let Azure Cognitive Search automatically pull in the data on a schedule. 4. Search an index - When submitting search requests to Azure Cognitive 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

Lab: Create an Azure Cognitive Search Service

Within the Azure Portal, click **New->Web + Mobile->Azure Cognitive Search**.

Once you click this, you'll have to fill out a few fields as you see fit. For this lab, a "Free" tier is sufficient.

Create New Azure Cognitive Search Service

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

Lab: Create an Azure Cognitive Search Index

An Index is the container for your data and is a similar concept to that of a SQL Server table. Like a table has rows, an Index has documents. Like a table that has fields, an Index has fields. 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 Cognitive Search by programatically pushing content or by using the Azure Cognitive Search Indexer (which can crawl common datastores for data).

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

Import Wizard

Within the Azure Cognitive Search blade you just created, click **Import Data->Data Source->Document DB**.

Import Wizard for DocDB

Once you click this, choose a name for the Cosmos DB datasource and choose the Cosmos DB account where your data resides as well as the cooresponding Container and Collections.

Click **OK**.

At this point Azure Cognitive Search will connect to your Cosmos DB container and analyze a few documents to identify a default schema for your Azure Cognitive Search Index. After this is complete, you can set the properties for the fields as needed by your application.

Update the Index name to: **images**

Update the Key to: **id** (which uniquely identifies each document)

Set all fields to be **Retrievable** (to allow the client to retrieve these fields when searched)

Set the fields **Tags, NumFaces, and Faces** to be **Filterable** (to allow the client to filter results based on these values)

Set the field **NumFaces** to be **Sortable** (to allow the client to sort the results based on the number of faces in the image)

Set the fields **Tags, NumFaces, and Faces** 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, Tags, and Faces** to be **Searchable** (to allow the client to do full text search over the text in these fields)

Configure Azure Cognitive Search Index

At this point we will configure the Azure Cognitive 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 Cognitive Search includes analyzers that are used in technologies like Bing and Office that have deep understanding of 56 languages.

Click the **Analyzer** tab and set the fields **Caption, Tags, and Faces** to use the **English-Microsoft** analyzer

Language Analyzers

For the final Index configuration step we will set the fields that will be used for type ahead, allowing the user to type parts of a word where Azure Cognitive Search will look for best matches in these fields

Click the **Suggester** tab and enter a Suggester Name: **sg** and choose **Tags and Faces** to be the fields to look for term suggestions

Search Suggestions

Click **OK** to complete the configuration of the Indexer. 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 Cognitive Search key field.

Click **OK, three times** to start the Indexer job that will start the importing of the data from the Cosmos DB database.

Configure Indexer

Query the Search Index

You should see a message pop up indicating that Indexing has started. If you wish to check the status of the Index, you can choose the "Indexes" option in the main Azure Cognitive Search blade.

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.

Search Explorer

Finish early? Try this extra credit lab:

Postman is a great tool that allows you to easily execute Azure Cognitive Search REST API calls and is a great debugging tool. You can take any query from the Azure Cognitive Search Explorer and along with an Azure Cognitive Search API key to be executed within Postman.

Download the Postman tool and install it.

After you have installed it, take a query from the Azure Cognitive Search explorer and paste it into Postman, choosing GET as the request type.

Click on Headers and enter the following parameters:

- Content Type: application/json
- api-key: [Enter your API key from the Azure Cognitive Search potal under the "Keys" section]

Choose send and you should see the data formatted in JSON format.

Try performing other searches using examples such as these.

LUIS

First, let's learn about Language Understand Intelligent Service (LUIS).

Now that we know what LUIS is, we'll want to plan our LUIS app. We'll be creating a bot 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 then create entities to model some parameters than are required to execute that action. For example, an intent for our PictureBot may be "SearchPics" and it triggers the Search service to look for photos, which requires a "facet" entity to know what to search for. You can see more examples for planning your app here.

Once we've thought out our app, we are ready to build and train it. These are the steps you will generally take when creating LUIS applications: 1. Add intents 2. Add utterances 3. Add entities 4. Improve performance using features 5. Train and test 6. Use active learning 7. Publish

Lab: Adding intelligence to your applications with LUIS

In the next lab, we're going to create our PictureBot. First, let's look at how we can use LUIS to add some natural language capabilities. LUIS allows you to map natural language utterances to intents. 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 and sign in using your Microsoft account. (This should be the same account that you used to create the Cognitive Services keys at the beginning of this lab.) You should be redirected to a list of your LUIS applications at https://www.luis.ai/applications. We will create a new LUIS app to support our bot.

> Fun Aside: Notice that there is also an "Import App" next to the "New App" button on the current page. After creating your LUIS application, you have the ability to export the entire app as JSON, and check it into source control. This is a recommended best practice so you can version your LUIS models as you version your code. An exported LUIS app may be re-imported using that "Import App" button. If you fall behind during the lab and want to cheat, you can click the "Import App" button and import the LUIS model.

From https://www.luis.ai/applications, click the "New App" button. Give it a name (I chose "PictureBotLuisModel") and set the Culture to "English". You can optionally provide a description. Click the dropdown to select an endpoint key to use, and if the LUIS key that you created on the Azure portal at the beginning of the workshop is there, select it (this option may not appear until you publish your app, so don't worry if you don't have any). Then click "Create".

LUIS New App

You will be taken to a Dashboard for your new app. The App Id is displayed; note that down for later as your **LUIS App ID**. Then click "Create an intent".

LUIS Dashboard

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 "Add intent" button.

Name the first intent "Greeting" and click "Save". Then give several examples of things the user might say when greeting the bot, pressing "Enter" after each one. After you have entered some utterances, click "Save".

LUIS Greeting Intent

Now 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 "Add custom entity". Give it an entity name "facet" and entity type "Simple". Then click "Save".

Add Facet Entity

Now click "Intents" in the left-hand sidebar and then click the yellow "Add Intent" button. Give it an intent name of "SearchPics" and then click "Save".

Now 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 a 6-month-old boy
- Please give me pics of 20-year-old women
- Show me beach pics
- I want to find dog photos
- Search for pictures of women indoors
- Show me pictures of girls looking happy
- I want to see pics of sad girls
- Show me happy baby pics

Now we have to teach LUIS how to pick out the search topic as the "facet" entity. Hover and click over the word (or drag to select a group of words) and then select the "facet" entity.

Labelling Entity

So the following list of utterances...

Add Facet Entity

...may become something like this when the facets are labelled.

Add Facet Entity

Don't forget to click "Save" when you are done!

Finally, 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.

Note too that there is one intent called "None". Random utterances that don't map to any of your intents may be mapped to "None". You are welcome to seed it with a few, like "Do you like peanut butter and jelly?"

Now we are ready to train our model. Click "Train & Test" in the left sidebar. Then click the train button. This builds a model to do utterance --> intent mapping with the training data you've provided.

Then click on "Publish App" in the left sidebar. If you have not already done so, select the endpoint key that you set up earlier, or follow the link to create a new key in your Azure account. You can leave the endpoint slot as "Production". Then click "Publish".

Publish LUIS App

Publishing creates an endpoint to call the LUIS model. The URL will be displayed.

Click on "Train & Test" in the left sidebar. Check the "Enable published model" box to have the calls go through the published endpoint rather than call the model directly. Try typing a few utterances and see the intents returned.

Test LUIS

Finish early? Try these extra credit tasks:

Create additional entities that can be leveraged by the "SearchPics" intent. For example, we know that our app determines age - try creating a prebuilt entity for age.

Explore using custom entities of entity type "List" to capture emotion and gender. See the example of emotion below.

Custom Emotion Entity with List

Note When you add more entities or features, don't forget to go to `Intents>Utterances` and confirm/add more utterances with the entities you add. Also, you will need to retrain and publish your model.

Building a Bot

We assume that you've had some exposure to the Bot Framework. If you have, great. If not, don't worry too much, you'll learn a lot in this section. We recommend completing this Microsoft Virtual Academy course and checking out the documentation.

Lab: Setting up for bot development

We will be developing a bot using the C# SDK. To get started, you need two things: 1. The Bot Framework project template, which you can download here. The file is called "Bot Application.zip" and you should save it into the \Documents\Visual Studio 2019\Templates\ProjectTemplates\Visual C#\ directory. Just drop the whole zipped file in there; no need to unzip.
2. Download the Bot Framework Emulator for testing your bot locally here. The emulator installs to `c:\Users\`*your-username*`\AppData\Local\botframework\app-3.5.27\botframework-emulator.exe`.

Lab: Creating a simple bot and running it

In Visual Studio, go to File --> New Project and create a Bot Application named "PictureBot".

New Bot Application

Browse around and examine the sample bot code, which is an echo bot that repeats back your message and its length in characters. In particular, **Note** + In **WebApiConfig.cs** under App_Start, the route template is api/{controller}/{id} where the id is optional. That is why we always call the bot's endpoint with api/messages appended at the end.
+ The **MessagesController.cs** under Controllers is therefore the entry point into your bot. Notice that a bot can respond to many different activity types, and sending a message will invoke the RootDialog.
+ In **RootDialog.cs** under Dialogs, "StartAsync" is the entry point which waits for a message from the user, and "MessageReceivedAsync" is the method that will handle the message once received and then wait for further messages. We can use "context.PostAsync" to send a message from the bot back to the user.

Right click on the solution and select **Manage NuGet Packages for this Solution**. Under installed, search for Microsoft.Bot.Builder, and update to the latest version.

Click F5 to run the sample code. NuGet should take care of downloading the appropriate dependencies.

The code will launch in your default web browser in a URL similar to http://localhost:3979/.

> Fun Aside: why this port number? It is set in your project properties. In your Solution Explorer, double-click "Properties" and select the "Web" tab. The Project Url is set in the "Servers" section.

Bot Project URL

Make sure your project is still running (hit F5 again if you stopped to look at the project properties) and launch the Bot Framework Emulator. (If you just installed it, it may not be indexed to show up in a search on your local machine, so remember that it installs to c:\Users\your-username\AppData\Local\botframework\app-3.5.27\botframework-emulator.exe.) Ensure that the Bot Url matches the port number that your code launched in above, and has api/messages appended to the end. Now you should be able to converse with the bot.

Bot Emulator

Lab: Update bot to use LUIS

Now we want to update our bot to use LUIS. We can do this by using the LuisDialog class.

In the **RootDialog.cs** file, add references to the following namespaces:

```csharp

using Microsoft.Bot.Builder.Luis; using Microsoft.Bot.Builder.Luis.Models;

```

Then, change the RootDialog class to derive from LuisDialog instead of IDialog. Then, give the class a LuisModel attribute with the LUIS App ID and LUIS key. (HINT: The LUIS App ID will have hyphens in it, and the LUIS key will not. If you can't find these values, go back to http://luis.ai. Click on your application, and the App ID is displayed right on the Dashboard page, as well as in the URL. Then click on "My keys" in the top sidebar to find your Endpoint Key in the list.)

```csharp

using System; using System.Threading.Tasks; using Microsoft.Bot.Builder.Dialogs; using Microsoft.Bot.Connector; using Microsoft.Bot.Builder.Luis; using Microsoft.Bot.Builder.Luis.Models;

namespace PictureBot.Dialogs { [LuisModel("96f65e22-7dcc-4f4d-a83a-d2aca5c72b24", "1234bb84eva3481a80c8a2a0fa2122f0")] [Serializable] public class RootDialog : LuisDialog {

```

> Fun Aside: You can use Autofac to dynamically load the LuisModel attribute on your class instead of hardcoding it, so it could be stored properly in a

configuration file. There is an example of this in the AlarmBot sample.

Next, delete the two existing methods in the class (StartAsync and MessageReceivedAsync). LuisDialog already has an implementation of StartAsync that will call the LUIS service and route to the appropriate method based on the response.

Finally, add a method for each intent. The corresponding method will be invoked for the highest-scoring intent. We will start by just displaying simple messages for each intent.

```csharp

    [LuisIntent("")]
    [LuisIntent("None")]
    public async Task None(IDialogContext context, LuisResult result)
    {
        await context.PostAsync("Hmmmm, I didn't understand that.  I'm
still learning!");
    }

    [LuisIntent("Greeting")]
    public async Task Greeting(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Hello!  I am a Photo Organization Bot.
I can search your photos, share your photos on Twitter, and order prints
of your photos.  You can ask me things like 'find pictures of food'.");
    }

    [LuisIntent("SearchPics")]
    public async Task SearchPics(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Searching for your pictures...");
    }

    [LuisIntent("OrderPic")]
    public async Task OrderPic(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Ordering your pictures...");
    }

    [LuisIntent("SharePic")]
    public async Task SharePic(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Posting your pictures to Twitter...");
    }

```

Now, let's run our code. Hit F5 to run in Visual Studio, and start up a new conversation in the Bot Framework Emulator. Try chatting with the bot, and ensure that you get the expected responses. If you get any unexpected results, note them down and we will revise LUIS.

Bot Test LUIS

In the above screenshot, I was expecting to get a different response when I said "order prints" to the bot. It looks like this was mapped to the "SearchPics" intent instead of the "OrderPic" intent. I can update my LUIS model by returning to http://luis.ai. Click on the appropriate application, and then click on "Intents" in the left sidebar. I could manually add this as a new utterance, or I could leverage the "suggested utterances" functionality in LUIS to improve my model. Click on the "SearchPics" intent (or the one to which your utterance was mis-labelled) and then click "Suggested utterances". Click the checkbox for the mis-labelled utterance, and then click "Reassign Intent" and select the correct intent.

LUIS Reassign Intent

For these changes to be picked up by your bot, you must re-train and re-publish your LUIS model. Click on "Publish App" in the left sidebar, and click the "Train" button and then the "Publish" button near the bottom. Then you can return to your bot in the emulator and try again.

> Fun Aside: The Suggested Utterances are extremely powerful. LUIS makes smart decisions about which utterances to surface. It chooses the ones that will help it improve the most to have manually labelled by a human-in-the-loop. For example, if the LUIS model predicted that a given utterance mapped to Intent1 with 47% confidence and predicted that it mapped to Intent2 with 48% confidence, that is a strong candidate to surface to a human to manually map, since the model is very close between two intents.

Now that we can use our LUIS model to figure out the user's intent, let's integrate Azure Cognitive Search to find our pictures.

Lab: Configure your bot for Azure Cognitive Search

First, we need to provide our bot with the relevant information to connect to an Azure Cognitive Search index. The best place to store connection information is in the configuration file.

Open Web.config and in the appSettings section, add the following:

```
xml <!-- Azure Cognitive Search Settings --> <add
key="SearchDialogsServiceName" value="" /> <add
key="SearchDialogsServiceKey" value="" /> <add
key="SearchDialogsIndexName" value="images" />
```

Set the value for the SearchDialogsServiceName to be the name of the Azure Cognitive Search Service that you created earlier. If needed, go back and look this up in the [Azure portal](#).

Set the value for the SearchDialogsServiceKey to be the key for this service. This can be found in the [Azure portal](#) under the Keys section for your Azure Cognitive Search. In the below screenshot, the SearchDialogsServiceName would be "aiimmersionsearch" and the SearchDialogsServiceKey would be "375...".

Azure Cognitive Search Settings

Lab: Update the bot to use Azure Cognitive Search

Now, let's update the bot to call Azure Cognitive Search. First, open Tools-->NuGet Package Manager-->Manage NuGet Packages for Solution. In the search box, type "Microsoft.Azure.Search". Select the corresponding library, check the box that indicates your project, and install it. It may install other dependencies as well. Under installed packages, you may also need to update the "Newtonsoft.Json" package.

Azure Cognitive Search NuGet

Right-click on your project in the Solution Explorer of Visual Studio, and select Add-->New Folder. Create a folder called "Models". Then right-click on the Models folder, and select Add-->Existing Item. Do this twice to add these two files under the Models folder (make sure to adjust your namespaces if necessary): 1. ImageMapper.cs 2. SearchHit.cs

Next, right-click on the Dialogs folder in the Solution Explorer of Visual Studio, and select Add-->Class. Call your class "SearchDialog.cs". Add the contents from here.

Finally, we need to update your RootDialog to call the SearchDialog. In RootDialog.cs in the Dialogs folder, update the SearchPics method and add these "ResumeAfter" methods:

~~~
```csharp

    [LuisIntent("SearchPics")]
    public async Task SearchPics(IDialogContext context, LuisResult
result)
    {
        // Check if LUIS has identified the search term that we should
look for.
        string facet = null;
        EntityRecommendation rec;
        if (result.TryFindEntity("facet", out rec)) facet = rec.Entity;

        // If we don't know what to search for (for example, the user
said
        // "find pictures" or "search" instead of "find pictures of x"),
        // then prompt for a search term.
        if (string.IsNullOrEmpty(facet))
        {
            PromptDialog.Text(context,
ResumeAfterSearchTopicClarification,
                "What kind of picture do you want to search for?");
        }
        else
        {
            await context.PostAsync("Searching pictures...");
            context.Call(new SearchDialog(facet),
ResumeAfterSearchDialog);
        }
    }
~~~

~~~
    private async Task
ResumeAfterSearchTopicClarification(IDialogContext context,
IAwaitable<string> result)
    {
        string searchTerm = await result;
        context.Call(new SearchDialog(searchTerm),
ResumeAfterSearchDialog);
    }

    private async Task ResumeAfterSearchDialog(IDialogContext context,
IAwaitable<object> result)
    {
        await context.PostAsync("Done searching pictures");
    }

```
~~~

Press F5 to run your bot again. In the Bot Emulator, try searching with "find dog pics" or "search for happiness photos". Ensure that you are seeing results when tags from your pictures are requested.

Lab: Regular expressions and scorable groups

There are a number of things that we can do to improve our bot. First of all, we may not want to call LUIS for a simple "hi" greeting, which the bot will get fairly frequently from its users. A simple regular expression could match this, and save us time (due to network latency) and money (due to cost of calling the LUIS service).

Also, as the complexity of our bot grows, and we are taking the user's input and using multiple services to interpret it, we need a process to manage that flow. For example, try regular expressions first, and if that doesn't match, call LUIS, and then perhaps we also drop down to try other services like QnA Maker and Azure Cognitive Search. A great way to manage this is ScorableGroups. ScorableGroups give you an attribute to impose an order on these service calls. In our code, let's impose an order of matching on regular expressions first, then calling LUIS for interpretation of utterances, and finally lowest priority is to drop down to a generic "I'm not sure what you mean" response.

To use ScorableGroups, your RootDialog will need to inherit from DispatchDialog instead of LuisDialog (but you can still have the LuisModel attribute on the class). You also will need a reference to Microsoft.Bot.Builder.Scorables (as well as others). So in your RootDialog.cs file, add:

```csharp

using Microsoft.Bot.Builder.Scorables; using System.Collections.Generic;

```

and change your class derivation to:

```csharp
```

```
public class RootDialog : DispatchDialog
```

Then let's add some new methods that match regular expressions as our first priority in ScorableGroup 0. Add the following at the beginning of your RootDialog class:

```csharp
    [RegexPattern("^hello")]
    [RegexPattern("^hi")]
    [ScorableGroup(0)]
    public async Task Hello(IDialogContext context, IActivity activity)
    {
        await context.PostAsync("Hello from RegEx!  I am a Photo
Organization Bot.  I can search your photos, share your photos on
Twitter, and order prints of your photos.  You can ask me things like
'find pictures of food'.");
    }

    [RegexPattern("^help")]
    [ScorableGroup(0)]
    public async Task Help(IDialogContext context, IActivity activity)
    {
        // Launch help dialog with button menu
        List<string> choices = new List<string>(new string[] { "Search
Pictures", "Share Picture", "Order Prints" });
        PromptDialog.Choice<string>(context, ResumeAfterChoice,
            new PromptOptions<string>("How can I help you?",
options:choices));
    }

    private async Task ResumeAfterChoice(IDialogContext context,
IAwaitable<string> result)
    {
        string choice = await result;

        switch (choice)
        {
            case "Search Pictures":
                PromptDialog.Text(context,
ResumeAfterSearchTopicClarification,
                    "What kind of picture do you want to search for?");
                break;
            case "Share Picture":
                await SharePic(context, null);
                break;
            case "Order Prints":
                await OrderPic(context, null);
                break;
            default:
                await context.PostAsync("I'm sorry. I didn't understand
you.");
                break;
        }
    }

```

This code will match on expressions from the user that start with "hi", "hello", and "help". Notice that when the user asks for help, we present him/her with a simple menu of buttons on the three core things our bot can do: search pictures, share pictures, and order prints.

> Fun Aside: One might argue that the user shouldn't have to type "help" to get a menu of clear options on what the bot can do; rather, this should be the default experience on first contact with the bot. **Discoverability** is one of the biggest challenges for bots - letting the users know what the bot is capable of doing. Good bot design principles can help.

Now we will call LUIS as our second attempt if no regular expression matches, in Scorable Group 1.

The "None" intent in LUIS means that the utterance didn't map to any intent. In this situation, we want to fall down to the next level of ScorableGroup. Modify your "None" method in the RootDialog class as follows:

```csharp

    [LuisIntent("")]
    [LuisIntent("None")]
    [ScorableGroup(1)]
    public async Task None(IDialogContext context, LuisResult result)
    {
        // Luis returned with "None" as the winning intent,
        // so drop down to next level of ScorableGroups.
        ContinueWithNextGroup();
    }

```

On the "Greeting" method, add a ScorableGroup attribute and add "from LUIS" to differentiate. When you run your code, try saying "hi" and "hello" (which should be caught by the RegEx match) and then say "greetings" or "hey there" (which may be caught by LUIS, depending on how you trained it). Note which method responds.

```csharp

    [LuisIntent("Greeting")]
    [ScorableGroup(1)]
    public async Task Greeting(IDialogContext context, LuisResult
result)
    {
        // Duplicate logic, for a teachable moment on Scorables.
        await context.PostAsync("Hello from LUIS!  I am a Photo
Organization Bot.  I can search your photos, share your photos on
Twitter, and order prints of your photos.  You can ask me things like
'find pictures of food'.");
    }

```

Then, add the ScorableGroup attribute to your "SearchPics" method and your "OrderPic" method.

```csharp

    [LuisIntent("SearchPics")]
    [ScorableGroup(1)]
    public async Task SearchPics(IDialogContext context, LuisResult
result)
    {
        ...
    }

    [LuisIntent("OrderPic")]
    [ScorableGroup(1)]
    public async Task OrderPic(IDialogContext context, LuisResult
result)
    {
        ...
    }

```

> Extra credit (to complete later): create an OrderDialog class in your "Dialogs" folder. Create a process for ordering prints with the bot using FormFlow. Your bot will need to collect the following information: Photo size (8x10, 5x7, wallet, etc.), number of prints, glossy or matte finish, user's phone number, and user's email.

You can update your "SharePic" method as well. This contains a little code to show how to do a prompt for a yes/no confirmation as well as setting the ScorableGroup. This code doesn't actually post a tweet because we didn't want to spend time getting everyone set up with Twitter developer accounts and such, but you are welcome to implement if you want.

```csharp

    [LuisIntent("SharePic")]
    [ScorableGroup(1)]
    public async Task SharePic(IDialogContext context, LuisResult
result)
    {
        PromptDialog.Confirm(context, AfterShareAsync,
            "Are you sure you want to tweet this picture?");
    }

    private async Task AfterShareAsync(IDialogContext context,
IAwaitable<bool> result)
    {
        if (result.GetAwaiter().GetResult() == true)
        {
            // Yes, share the picture.
            await context.PostAsync("Posting tweet.");
        }
        else
        {
            // No, don't share the picture.
            await context.PostAsync("OK, I won't share it.");
        }
    }

```

Finally, add a default handler if none of the above services were able to understand. This ScorableGroup needs an explicit MethodBind because it is not decorated with a LuisIntent or RegexPattern attribute (which include a MethodBind).

```csharp

    // Since none of the scorables in previous group won, the dialog
sends a help message.
    [MethodBind]
    [ScorableGroup(2)]
    public async Task Default(IDialogContext context, IActivity
activity)
    {
        await context.PostAsync("I'm sorry. I didn't understand you.");
        await context.PostAsync("You can tell me to find photos, tweet
them, and order prints.  Here is an example: \"find pictures of
food\".");
    }

```

Hit F5 to run your bot and test it in the Bot Emulator.

Lab: Publish your bot

A bot created using the Microsoft Bot can be hosted at any publicly-accessible URL. For the purposes of this lab, we will host our bot in an Azure website/app service.

In the Solution Explorer in Visual Studio, right-click on your Bot Application project and select "Publish". This will launch a wizard to help you publish your bot to Azure.

Select the publish target of "Microsoft Azure App Service".

Publish Bot to Azure App Service

On the App Service screen, select the appropriate subscription and click "New". Then enter an API app name, subscription, the same resource group that you've been using thus far, and an app service plan.

Create App Service

Finally, you will see the Web Deploy settings, and can click "Publish". The output window in Visual Studio will show the deployment process. Then, your bot will be hosted at a URL like http://testpicturebot.azurewebsites.net/, where "testpicturebot" is the App Service API app name.

Lab: Register your bot with the Bot Connector

Now, go to a web browser and navigate to http://dev.botframework.com. Click Register a bot. Fill out your bot's name, handle, and description. Your messaging endpoint will be your Azure website URL with "api/messages" appended to the end, like https://testpicturebot.azurewebsites.net/api/messages.

Bot Registration

Then click the button to create a Microsoft App ID and password. This is your Bot App ID and password that you will need in your Web.config. Store your Bot app name, app ID, and app password in a safe place! Once you click "OK" on the password, there is no way to get back to it. Then click "Finish and go back to Bot Framework".

Bot Generate App Name, ID, and Password

On the bot registration page, your app ID should have been automatically filled in. You can optionally add an AppInsights instrumentation key for logging from your bot. Check the box if you agree with the terms of service and click "Register".

You are then taken to your bot's dashboard page, with a URL like https://dev.botframework.com/bots?id=TestPictureBot but with your own bot name. This is where we can enable various channels. Two channels, Skype and Web Chat, are enabled automatically.

Finally, you need to update your bot with its registration information. Return to Visual Studio and open Web.config. Update the BotId with the App Name, the MicrosoftAppId with the App ID, and the MicrosoftAppPassword with the App Password that you got from the bot registration site.

```xml
<add key="BotId" value="TestPictureBot" />
<add key="MicrosoftAppId" value="95b76ae6-8643-4d94-b8a1-916d9f753ab0"
/>
<add key="MicrosoftAppPassword" value="kC200000000000000000000" />
```

Rebuild your project, and then right-click on the project in the Solution Explorer and select "Publish" again. Your settings should be remembered from last time, so you can just hit "Publish".

> Getting an error that directs you to your MicrosoftAppPassword? Because it's in XML, if your key contains "&", "<", ">", "'", or '"', you will need to replace those symbols with their respective escape facilities: "&", "<", ">", "'", """.

Now you can navigate back to your bot's dashboard (something like https://dev.botframework.com/bots?id=TestPictureBot). Try talking to it in the Chat window. The carousel may look different in Web Chat than the emulator. There is a

great tool called the Channel Inspector to see the user experience of various controls in the different channels at https://docs.botframework.com/en-us/channel-inspector/channels/Skype/#navtitle.
From your bot's dashboard, you can add other channels, and try out your bot in Skype, Facebook Messenger, or Slack. Simply click the "Add" button to the right of the channel name on your bot's dashboard, and follow the instructions.

Finish early? Try this extra credit lab:

Try experimenting with more advanced Azure Cognitive Search queries. Add term-boosting by extending your LUIS model to recognize entities like *"find happy people"*, mapping "happy" to "happiness" (the emotion returned from Cognitive Services), and turning those into boosted queries using Term Boosting.

Lab Completion

In this lab we covered creating an intelligent bot from end-to-end using the Microsoft Bot Framework, Azure Cognitive Search and several Cognitive Services.

You should have learned: - How to weave intelligent services into your applications - How to implement Azure Cognitive Search features to provide a positive search experience inside an application - How to configure an Azure Cognitive Search service to extend your data to enable full-text, language-aware search - How to build, train and publish a LUIS model to help your bot communicate effectively - How to build an intelligent bot using Microsoft Bot Framework that leverages LUIS and Azure Cognitive Search - How to call various Cognitive Services APIs (specifically Computer Vision, Face, Emotion and LUIS) in .NET applications

Resources for future projects/learning: - Azure Bot Services documentation - Azure Cognitive Search documentation - Azure Bot Builder Samples - Azure Cognitive Search Samples - LUIS documentation - LUIS Sample

Developing Intelligent Applications with LUIS and Azure Search

This hands-on lab guides you through creating an intelligent bot from end-to-end using the Microsoft Bot Framework, Azure Search, and Microsoft's Language Understanding Intelligent Service (LUIS).

Objectives

In this workshop, you will: - Understand how to implement Azure Search features to provide a positive search experience inside applications - Configure an Azure Search service to extend your data to enable full-text, language-aware search - Build, train and publish a LUIS model to help your bot communicate effectively - Build an intelligent bot using Microsoft Bot Framework that leverages LUIS and Azure Search

While there is a focus on LUIS and Azure Search, you will also leverage the following technologies:

- Data Science Virtual Machine (DSVM)
- Windows 10 SDK (UWP)
- CosmosDB
- Azure Storage
- Visual Studio

Prerequisites

This workshop is meant for an AI Developer on Azure. Since this is a short workshop, there are certain things you need before you arrive.

Firstly, you should have experience with Visual Studio. We will be using it for everything we are building in the workshop, so you should be familiar with how to use it to create applications. Additionally, this is not a class where we teach you how to code or develop applications. We assume you know how to code in C# (you can learn here), but you do not know how to implement advanced Search and NLP (natural language processing) solutions.

Secondly, you should have some experience developing bots with Microsoft's Bot Framework. We won't spend a lot of time discussing how to design them or how dialogs work. If you are not familiar with the Bot Framework, you should take this Microsoft Virtual Academy course prior to attending the workshop.

Thirdly, you should have experience with the portal and be able to create resources (and spend money) on Azure. We will not be providing Azure passes for this workshop.

> **Note** This workshop was developed and tested on a Data Science Virtual Machine (DSVM) with Visual Studio Community Version 15.4.0

Introduction

We're going to build an end-to-end scenario that allows you to pull in your own pictures, use Cognitive Services to find objects and people in the images, figure out how those people are feeling, and store all of that data into a NoSQL Store (DocumentDB). We'll use that NoSQL Store to populate an Azure Search index, and then build a Bot Framework bot using LUIS to allow easy, targeted querying.

> **Note** This lab is a continuation of `lab01.1-pcl_and_cognitive_services`; we will skip pulling in pictures, using Cognitive Services to determine information about the images, and storing the data in DocumentDB. The only thing we will use from that lab is DocumentDB to populate our search index. If you have completed `lab01.1-pcl_and_cognitive_services`, you can optionally use your DocumentDB connection string instead of the one provided.

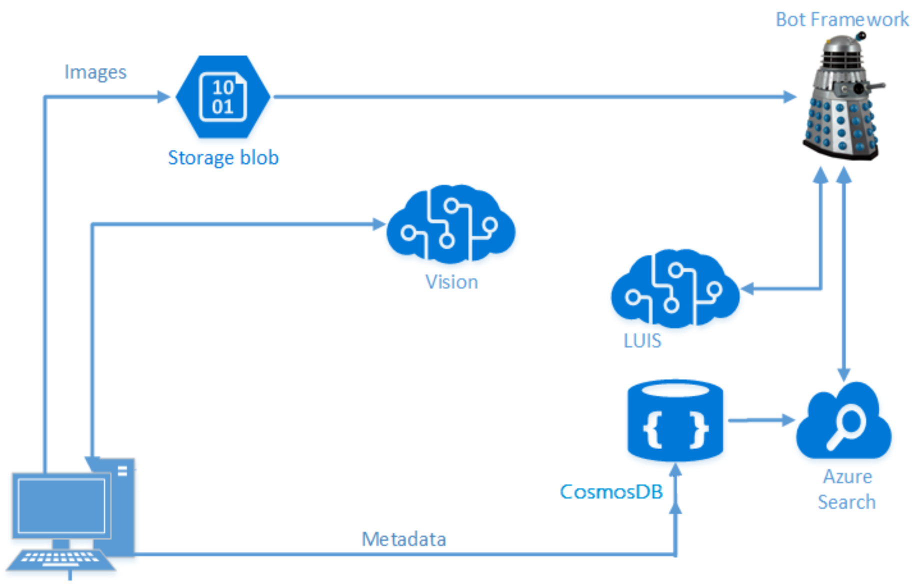
Architecture

In `lab01.1-pcl_and_cognitive_services`, we built a simple C# application that allows you to ingest pictures from your local drive, then invoke several different Cognitive Services to gather data on those images:

- Computer Vision: We use this to grab tags and a description
- Face: We use this to grab faces and their details from each image
- Emotion: We use this to pull emotion scores from each face in the image

Once we had this data, we processed it and stored all the information needed in DocumentDB, our NoSQL PaaS offering.

Now that we have it in DocumentDB, we'll build an Azure Search Index on top of it (Azure Search is our PaaS offering for faceted, fault-tolerant search - think Elastic Search without the management overhead). We'll show you how to query your data, and then build a Bot Framework bot to query it. Finally, we'll extend this bot with LUIS to automatically derive intent from your queries and use those to direct your searches intelligently.



This lab was modified from this Cognitive Services Tutorial.

Navigating the GitHub

There are several directories in the resources folder:

- **assets**: This contains all of the images for the lab manual. You can ignore this folder.
- **code**: In here, there are several directories that we will use:
  - **LUIS**: Here you will find the LUIS model for the PictureBot. You will create your own, but if you fall behind or want to test out a different LUIS model, you can use the .json file to import this LUIS app.
  - **Models**: These classes will be used when we add search to our PictureBot.
  - **Finished-PictureBot**: Here there is the finished PictureBot.sln that is for the latter sections of the workshop, where we integrate LUIS and our Search Index into the Bot Framework. If you fall behind or get stuck, you can use this.

You need Visual Studio to run these labs, but if you have already deployed a Windows Data Science Virtual Machine for one of the workshops, you could use that.

Collecting the Keys

Over the course of this lab, we will collect various keys. It is recommended that you save all of them in a text file, so you can easily access them throughout the workshop.

> *Keys* - LUIS API: - Cosmos DB Connection String: - Azure Search Name: - Azure Search Key: - Bot Framework App Name: - Bot Framework App ID: - Bot Framework App Password:

Navigating the Labs

This workshop has been broken down into five sections: - 1_AzureSearch: Here you will learn about Azure Search and create an index - 2_LUIS: We'll build a LUIS model to enhance the language understanding of our bot (which you'll build in the next lab) - 3_Bot: Learn how to put everything together using the Bot Framework - 4_Bot_Enhancements: We'll finish by enhancing our bot with regular expressions and publishing it with the Bot Connector - 5_Challenge_and_Closing: If you get through all the labs, try this challenge. You will also find a summary of what you've done and where to learn more.

Continue to 1_AzureSearch

1_AzureSearch:

Estimated Time: 20-30 minutes

Azure Search

Azure Search is a search-as-a-service solution allowing developers to incorporate great search experiences into applications without managing infrastructure or needing to become search experts.

Developers look for PaaS services in Azure to achieve better and faster results in their apps. While search is a key to many types of applications, web search engines have set the bar high for search. Users expect: instant results, auto-complete as they type, highlighting hits within the results, great ranking, and the ability to understand what they are looking for, even if they spell it incorrectly or include extra words.

Search is a hard and rarely a core expertise area. From an infrastructure standpoint, it needs to have high availability, durability, scale, and operations. From a functionality standpoint, it needs to have ranking, language support, and geospatial capabilities.

Example of Search Requirements

The example above illustrates some of the components users are expecting in their search experience. Azure Search can accomplish these user experience features, along with giving you monitoring and reporting, simple scoring, and tools for prototyping and inspection.

Typical Workflow: 1. Provision service - You can create or provision an Azure Search service from the portal or with PowerShell. 2. 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. 3. Index 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. 4. 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

Lab: Create an Azure Search Service

Within the Azure Portal, click **Create a resource->Web + Mobile->Azure Search**.

Once you click this, you'll have to fill out a few fields as you see fit. For this lab, the "F0" 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 completed `lab01.1-pcl_and_cognitive_services`, you can just use that Resource Group.

Create New Azure Search Service

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

Lab: 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 reflect 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 container.

Import Wizard

Within the Azure Search blade you just created, click **Import Data->Data Source->Document DB**.

Import Wizard for DocDB

Once you click this, choose a name for the Cosmos DB data source. If you completed the previous lab, `lab01.1-pcl_and_cognitive_services`, 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, select "Or input a connection string" and paste in `AccountEndpoint=https://timedcosmosdb.documents.azure.com:443/;AccountKey=0aRt6JVgbf9KafBxRVuDMNfAj9YoSBbmpICdJ41N5CwHcjuMcVk7jWDBcu4BxbTitLR1zteauQsnF1Tgqs1A3g==;`. For both, the Database should be "images" and the Collection should be "metadata".

Click **OK**.

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.

> **Note** You may see a warning that "_ts" fields are not valid field names. You can ignore this for our labs, but you can read more about it here.

Update the Index name to: **images**

Update the Key to: **id** (which uniquely identifies each document)

Set all fields to be **Retrievable** (to allow the client to retrieve these fields when searched)

Set the fields **Tags, NumFaces, and Faces** to be **Filterable** (to allow the client to filter results based on these values)

Set the field **NumFaces** to be **Sortable** (to allow the client to sort the results based on the number of faces in the image)

Set the fields **Tags, NumFaces, and Faces** 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, Tags, and Faces** to be **Searchable** (to allow the client to do full text search over the text in these fields)

Configure Azure Search Index

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 56 languages.

Click the **Analyzer** tab and set the fields **Caption, Tags, and Faces** to use the **English-Microsoft** analyzer.

Language Analyzers

For the final Index configuration step, we will 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.

Click the **Suggester** tab and enter a Suggester Name: **sg** and choose **Tags and Faces** to be the fields to look for term suggestions

Search Suggestions

Click **OK** to complete the configuration of the Indexer. 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, three times** to start the Indexer job that will start the importing of the data from the Cosmos DB database.

Configure Indexer

Query the Search Index

You should see a message pop up indicating that Indexing has started. If you wish to check the status of the Index, you can choose the "Indexes" option in the main Azure Search blade.

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?

Search Explorer

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.

Finish early? Try this extra credit lab:

Postman is a great tool that allows you to easily execute Azure Search REST API calls and is a great debugging tool. You can take any query from the Azure Search Explorer and along with an Azure Search API key to be executed within Postman.

Download the Postman tool and install it.

After you have installed it, take a query from the Azure Search explorer and paste it into Postman, choosing GET as the request type.

Click on Headers and enter the following parameters:

- Content Type: application/json
- api-key: [Enter your API key from the Azure Search portal under the "Keys" section]

Choose send and you should see the data formatted in JSON format.

Try performing other searches using examples such as these.

Continue to 2_LUIS

Back to [README

2_Luis:

Estimated Time: 20-30 minutes

LUIS

First, let's learn about Microsoft's Language Understand Intelligent Service (LUIS).

Now that we know what LUIS is, we'll want to plan our LUIS app. We'll be creating a bot 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 then create entities to model some parameters than are required to execute that action. For example, an intent for our PictureBot may be "SearchPics" and it triggers the Search service to look for photos, which requires a "facet" entity to know what to search for. You can see more examples for planning your app here.

Once we've thought out our app, we are ready to build and train it. These are the steps you will generally take when creating LUIS applications: 1. Add intents 2. Add utterances 3. Add entities 4. Improve performance using features 5. Train and test 6. Use active learning 7. Publish

Lab: Creating the LUIS service in the portal

In the Portal, hit **Create a resource** and then enter **LUIS** in the search box and choose **Language Understanding Intelligent Service**:

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. The free tier is sufficient for this lab. Since LUIS stores images internally at Microsoft (in a secure fashion), to help improve future Cognitive Services offerings, you'll need to check the box to confirm you're ok with this.

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.

Cognitive API Key

Lab: Adding intelligence to your applications with LUIS

In the next lab, we will create our PictureBot. First, 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 and sign in using your Microsoft account. (This should be the same account that you used to create the LUIS key in the previous section).

You should be redirected to a list of your LUIS applications at https://www.luis.ai/applications. We will create a new LUIS app to support our bot.

> Fun Aside: Notice that there is also an "Import App" next to the "New App" button on the current page. After creating your LUIS application, you have the ability to export the entire app as JSON and check it into source control. This is a recommended best practice, so you can version your LUIS models as you version your code. An exported LUIS app may be re-imported using that "Import App" button. If you fall behind during the lab and want to cheat, you can click the "Import App" button and import the LUIS model.

From https://www.luis.ai/applications, click the "New App" button. Give it a name (I chose "PictureBotLuisModel") and set the Culture to "English". You can optionally provide a description. Then click "Create".

LUIS New App

You will be taken to a Dashboard for your new app. The App Id is displayed; note that down for later as your **LUIS App ID**. Then click "Create an intent".

LUIS Dashboard

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 "Add intent" button.

Name the first intent "Greeting" and click "Save". Then give several examples of things the user might say when greeting the bot, pressing "Enter" after each one. After you have entered some utterances, click "Save".

LUIS Greeting Intent

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 "Add custom entity". Give it an entity name "facet" and entity type "Simple". Then click "Save".

Add Facet Entity

Next, click "Intents" in the left-hand sidebar and then click the yellow "Add Intent" button. Give it an intent name of "SearchPics" and then click "Save".

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 a 6-month-old boy
- Please give me pics of 20-year-old women
- Show me beach pics
- I want to find dog photos
- Search for pictures of women indoors
- Show me pictures of girls looking happy
- I want to see pics of sad girls
- 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 over the word (or drag to select a group of words) and then select the "facet" entity.

Labelling Entity

So the following list of utterances...

Add Facet Entity

...may become something like this when the facets are labeled.

Add Facet Entity

Don't forget to click "Save" when you are done!

Finally, 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.

Note too that there is one intent called "None". Random utterances that don't map to any of your intents may be mapped to "None".

We are now ready to train our model. Click "Train & Test" in the left sidebar. Then click the train button. 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.

Then click on "Publish App" in the left sidebar. You have several options when you publish your app, including enabling verbose endpoint response or Bing spell checker. If you have not already done so, select the endpoint key that you set up earlier, or follow the link to add a key from your Azure account. You can leave the endpoint slot as "Production". Then click "Publish".

Publish LUIS App

Publishing creates an endpoint to call the LUIS model. The URL will be displayed, which will be explained in a later lab.

Click on "Train & Test" in the left sidebar. Check the "Enable published model" box to have the calls go through the published endpoint rather than call the model directly. Try typing a few utterances and see the intents returned.

> Unfortunately, there is a bug open with "Enable published model", and it only works in Chrome. You can either download Chrome and try this, or skip it but remember that it's not enabled.

Test LUIS

You can also test your published endpoint in a browser. Copy the URL, then replace the `{YOUR-KEY-HERE}` with one of the keys listed in the Key String column for the resource you want to use. To open this URL in your browser, set the URL parameter `&q` to your test query. For example, append `&q=Find pictures of dogs` to your URL, and then press Enter. The browser displays the JSON response of your HTTP endpoint.

Finish early? Try these extra credit tasks:

Create additional entities that can be leveraged by the "SearchPics" intent. For example, we know that our app determines age - try creating a prebuilt entity for age.

Explore using custom entities of entity type "List" to capture emotion and gender. See the example of emotion below.

Custom Emotion Entity with List

> **Note** When you add more entities or features, don't forget to go to **Intents>Utterances** and confirm/add more utterances with the entities you add. Also, you will need to retrain and publish your model.

Continue to 3_Bot

Back to [README

3_Bot:

Estimated Time: 30-40 minutes

Building a Bot

We assume that you've had some exposure to the Bot Framework. If you have, great. If not, don't worry too much, you'll learn a lot in this section. We recommend completing this Microsoft Virtual Academy course and checking out the documentation.

Lab: Setting up for bot development

We will be developing a bot using the C# SDK. To get started, you need two things: 1. The Bot Framework project template, which you can download here. The file is called "Bot Application.zip" and you should save it into the \Documents\Visual Studio 2019\Templates\ProjectTemplates\Visual C#\ directory. Just drop the whole zipped file in there; no need to unzip.
2. Download the Bot Framework Emulator for testing your bot locally here. The emulator installs to `c:\Users\`*your-username*`\AppData\Local\botframework\app-3.5.33\botframework-emulator.exe`.

Lab: Creating a simple bot and running it

In Visual Studio, go to File --> New Project and create a Bot Application named "PictureBot". Make sure you name it "PictureBot" or you may have issues later on.

New Bot Application

> The rest of the **Creating a simple bot and running it** lab is optional. Per the prerequisites, you should have experience working with the Bot Framework. You can hit F5 to confirm it builds correctly, and move on to the next lab.

Browse around and examine the sample bot code, which is an echo bot that repeats back your message and its length in characters. In particular, **Note** + In **WebApiConfig.cs** under App_Start, the route template is api/{controller}/{id} where the id is optional. That is why we always call the bot's endpoint with api/messages appended at the end.
+ The **MessagesController.cs** under Controllers is therefore the entry point into your bot. Notice that a bot can respond to many different activity types, and sending a message will invoke the RootDialog.
+ In **RootDialog.cs** under Dialogs, "StartAsync" is the entry point which waits for a message from the user, and "MessageReceivedAsync" is the method that will handle the message once received and then wait for further messages. We can use "context.PostAsync" to send a message from the bot back to the user.

Click F5 to run the sample code. NuGet should take care of downloading the appropriate dependencies.

The code will launch in your default web browser in a URL similar to http://localhost:3979/.

> Fun Aside: why this port number? It is set in your project properties. In your Solution Explorer, double-click "Properties" and select the "Web" tab. The Project URL is set in the "Servers" section.

Bot Project URL

Make sure your project is still running (hit F5 again if you stopped to look at the project properties) and launch the Bot Framework Emulator. (If you just installed it, it may not be indexed to show up in a search on your local machine, so remember that it installs to c:\Users\your-username\AppData\Local\botframework\app-3.5.27\botframework-emulator.exe.) Ensure that the Bot URL matches the port number that your code launched in above, and has api/messages appended to the end. You should be able to converse with the bot.

Bot Emulator

Lab: Update bot to use LUIS

We have to update our bot in order to use LUIS. We can do this by using the [LuisDialog class](#).

In the **RootDialog.cs** file, add references to the following namespaces:

```csharp

using Microsoft.Bot.Builder.Luis; using Microsoft.Bot.Builder.Luis.Models;

```

Then, change the RootDialog class to derive from LuisDialog

Table of Contents

Lab 05-01: QnA Maker	1
DebugActivityLogger	1
Global.asax.cs	1
2_File_Logger	1
3_SQL_Logger	1
Lab 05-02: QnA Maker Setup	1
Lab 06-01: Implement LUIS	1
Lab 07-01: Integrate LUIS	1
Lab 07-02: Integrate LUIS with Bots	1
code/Finished	1
Lab 08-01: Detect Language	1
Lab 08-02: Detect Language	1
Lab 09-01: Test Bot DirectLine	1
Lab 09-02: Test Bots Directline	1
Bot's Web.config	1
client's App.config	1
Program.cs	1
2_Unit_Testing_Bots	1
3_Direct_Line_Testing	1