



Bot in a Day



Version 5.18

Contents

Lab Prerequisites.....	4
Lab Structure	7
Overview	8
Lab 1 – Develop bots with the Bot Builder.....	9
Create your bot	10
Test your bot.....	15
Lab 1.1 – Develop bots by leveraging QnA Maker	18
Creating and Training our Knowledge Base using QnA Maker Service.....	19
Creating and Connecting a Bot to QnA Maker Service	26
Lab 1.2 – QnA Maker with rich cards in .NET.....	36
Adding delimiters to QnA and Rich Card Creation	37
Extra Exercise.....	45

Lab 2 – Interacting with the user – Multiple Dialogs	46
Invoking multiples dialogs	47
Create a simple game	54
Lab 2.1 - Interacting with the user – Adaptive Cards	63
Creating Adaptive Cards and Prompts	64
Extra Exercise.....	75
Reading for Scorable Dialogs.....	76
Lab 3 – Building Forms with FormFlow and Language Understanding with Microsoft Cognitive Services	77
Building a Form with FormFlow	78
Lab 3.1 – Interpreting Emotional Sentiment using Language Understanding (LUIS) with Cognitive Services.....	83
Create your LUIS app	85
Connect LUIS to your bot.....	93

Lab Prerequisites

This workshop is intended for emerging developers looking to learn how to develop chat bots using conversational AI on Microsoft Azure. Since this is only a short workshop, there are certain things you need before you arrive.

Firstly, you should have some previous exposure to 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.

Read an overview, get started with, and take tutorials of Visual Studio here: <https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide>

Additionally, this is not a class where we teach you how to code or develop applications. We assume you have some familiarity with C#, but you do not know how to implement solutions with Microsoft Cognitive Services.

You can learn foundational C# skills here: https://mva.microsoft.com/en-us/training-courses/c-fundamentals-for-absolute-beginners-16169?l=Lvld4EQIC_2706218949

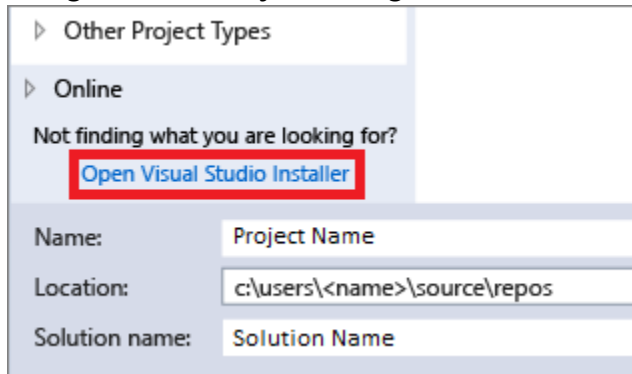
Finally, you should have little to no experience developing bots with the [Microsoft Bot Framework](#) and the [Bot Builder SDK for .NET](#). We will spend a lot of time discussing how to leverage [Microsoft Cognitive Services](#) to help you infuse AI into your bot, as well as practical approaches to interacting with the user.

Finally, before arriving at the workshop, we expect you to have done the following:

- Bring a laptop PC with at least 2-cores (2Ghz) and 4Gb RAM running on one of the following versions of Windows: 7 (SP1), 8.1, 10, or Windows Server 2012 R2 Standard or Datacenter
- Have a [Microsoft account](#) (Outlook, Live, Hotmail, MSN).
- Create or have an [Azure subscription](#). For identification purposes, it will require a credit card to be associated with your account. However, the services we will use for the purposes of this workshop will only require the free plan pricing tier. You will not be charged for the use of this service.

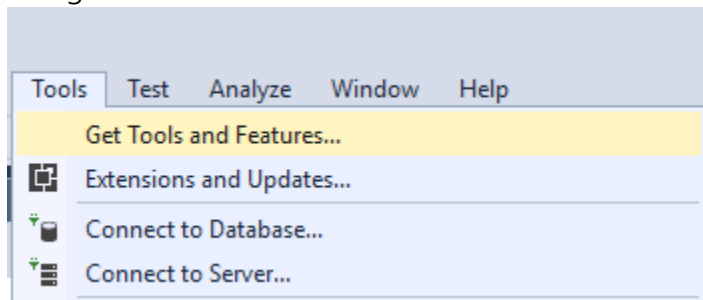
(continued on the next page)

- Installed [Visual Studio](#) 2015 or later (Community, Professional, or Enterprise edition)
 - Add the ASP.NET and web development workload by:
 - Using the New Project dialog box

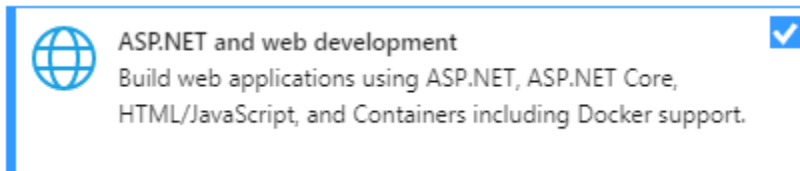


OR

- Using the Tools menu bar



- The Visual Studio Installer launches, select the workload, and then choose Modify



(continued on the next page)

- Installed [Internet Information Services \(IIS\) 10.0 Express](#)
- Installed the [Bot Framework Emulator](#)
- Downloaded the [Bot Application](#), [Bot Controller](#) and [Bot Dialog](#) .zip files.
 - Install the project template by copying Bot Application.zip to your Visual Studio project templates directory. The project templates directory is typically located at:
%USERPROFILE%\Documents\Visual Studio 2017\Templates\ProjectTemplates\Visual C#\
 - Install the item templates by copying Bot Controller.zip and Bot Dialog.zip to your item templates directory. The item templates directory is typically located at:
%USERPROFILE%\Documents\Visual Studio 2017\Templates\ItemTemplates\Visual C#\

Lab Structure

This document has 3 main sections:

- Lab 1
- Lab 2
- Lab 3

This manual is in table format. On the left panel are the steps the user needs to follow and in the right panel are screenshots to provide a visual aid for the users. In the screenshots, sections are highlighted with red boxes to highlight the action/area the user needs to focus on

NOTE: This lab is using material adapted from Microsoft Open Source GitHub Repositories hosted and presented by Microsoft on Developing and Deploying Intelligent Chat Bots. The material is intended to assist you in learning and is filed under the Creative Commons license. This lab has been created and provided by Neal Analytics, LLC—and exclusive Microsoft partner. Visit their site to learn more about their services: <https://www.nealanalytics.com>

Microsoft and any contributors grant you a license to the Microsoft documentation and other content in this repository under the [Creative Commons Attribution 4.0 International Public License](<https://creativecommons.org/licenses/by/4.0/legalcode>), see the LICENSE file, and grant you a license to any code in the repository under the [MIT License](<https://opensource.org/licenses/MIT>), see the LICENSE-CODE file.

Microsoft, Azure, and/or other Microsoft products and services referenced in the documentation may be either trademarks or registered trademarks of Microsoft in the United States and/or other countries. The licenses for this project do not grant you the rights to use any Microsoft names, logos, or trademarks.

Microsoft's general trademark guidelines can be found at <http://go.microsoft.com/fwlink/?LinkID=254653>.

Privacy information can be found at <https://privacy.microsoft.com/>

Microsoft and any contributors reserve all other rights, whether under their respective copyrights, patents, or trademarks, whether by implication, estoppel or otherwise.

Overview

Introduction

Today you will be learning how to quickly build and test a simple bot by following instructions in a step-by-step tutorial. This is an introductory to intermediate course intended to learn key concepts and design principles by leveraging the Bot Builder SDK and the Bot Builder & Bot Framework Samples found here: <https://github.com/Microsoft/BotBuilder-Samples>

The student is expected to have experience in using Visual Studio and a fundamental knowledge of .NET framework as well as some experience with the C# (C Sharp) programming language.

Code Samples

The code you will use today is a sample bot adapted from Microsoft bot samples designed to introduce key concepts and basic design principles.

Course Outline

Agenda *(times are approximate and will be fluid with the class)*

Morning

- 09:00 AM – 10:00 AM - Introduction to Bot Framework & QnA Maker
- 10:00 AM – 10:30 AM - Partner Overview & Demo
- 10:30 AM – 11:30 AM - Lab 1 & 1.1 - Develop bots with Bot Builder and QnA Maker
- 11:30 AM – 12:00 PM - Lab 1.2 - Interacting with the User – Adding Rich Cards
- 12:00 PM – 12:30 PM - Lunch

Afternoon

- 12:30 PM – 1:00 PM - Interacting with the user – Dialogs, Cards, and FormFlow
- 1:00 PM – 02:30 PM - Lab 2 & 2.1 – Multiple Dialogs & Adaptive Cards
- 02:30 PM – 03:00 PM - Introduction to Cognitive Services & LUIS
- 03:00 PM – 04:30 PM - Lab 3 & 3.1 – Building Forms and Language Understanding
- 04:30 PM – 05:00 PM - Q&A

Lab 1 – Develop bots with the Bot Builder

In this section, you will learn how to leverage the Bot Builder SDK, libraries, samples, and tools to help you build and develop bots. When you build a bot with Bot Service, your bot is backed by the Bot Builder SDK. You can also use the Bot Builder SDK to create a bot from scratch using C# (or Node.js). Bot Builder includes the Bot Framework Emulator for testing your bots and the Channel Inspector for previewing your bot's user experience on different channels.

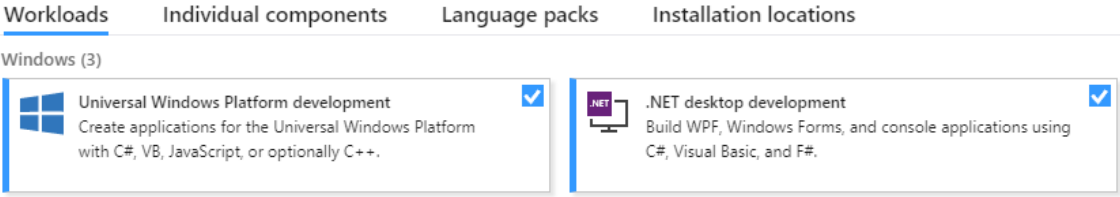
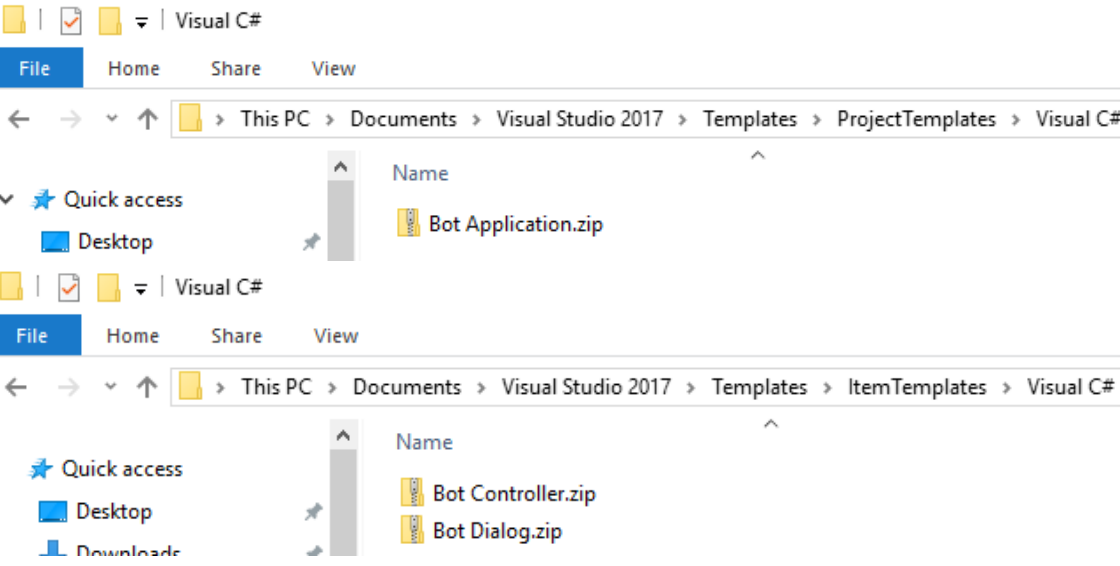
The Bot Builder SDK for .NET is an easy-to-use framework for developing bots using Visual Studio and Windows. The SDK leverages C# to provide a familiar way for .NET developers to create powerful bots.

This lab walks you through building a bot by using the Bot Application template and the Bot Builder SDK for .NET, and then testing it with the Bot Framework Emulator.

Learning Objectives

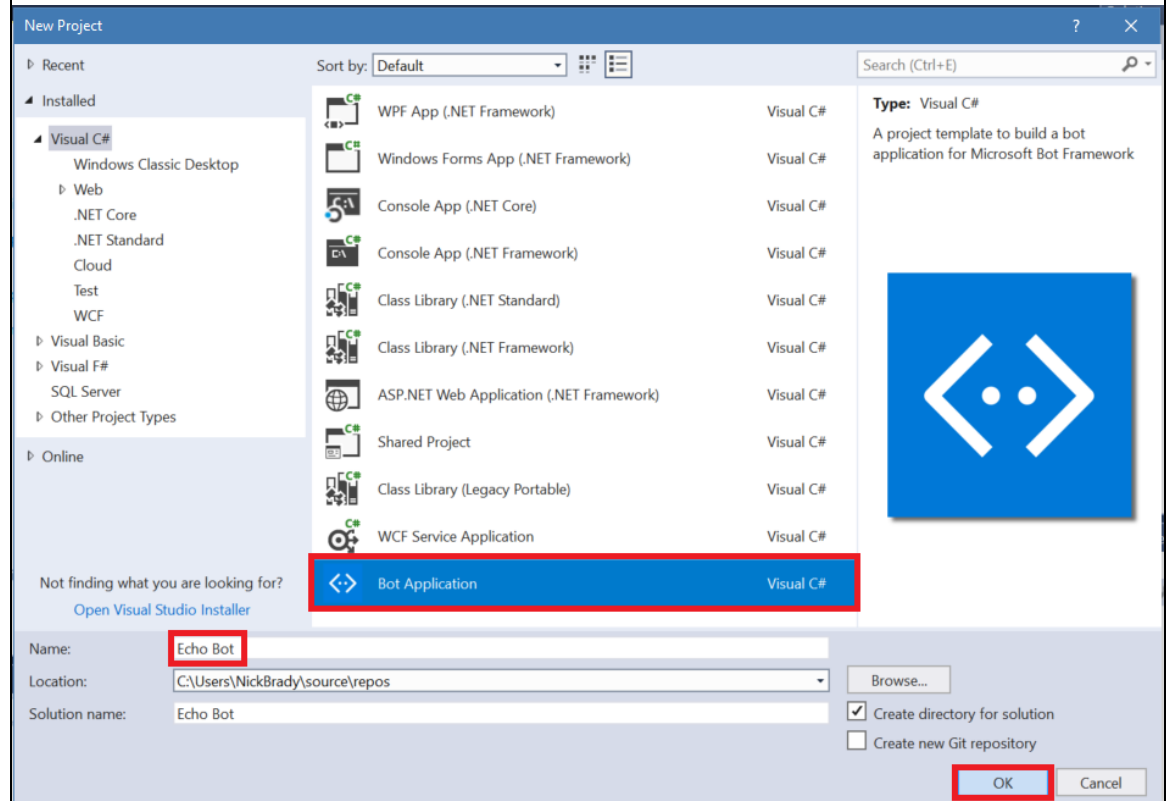
Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft's Bot Framework.

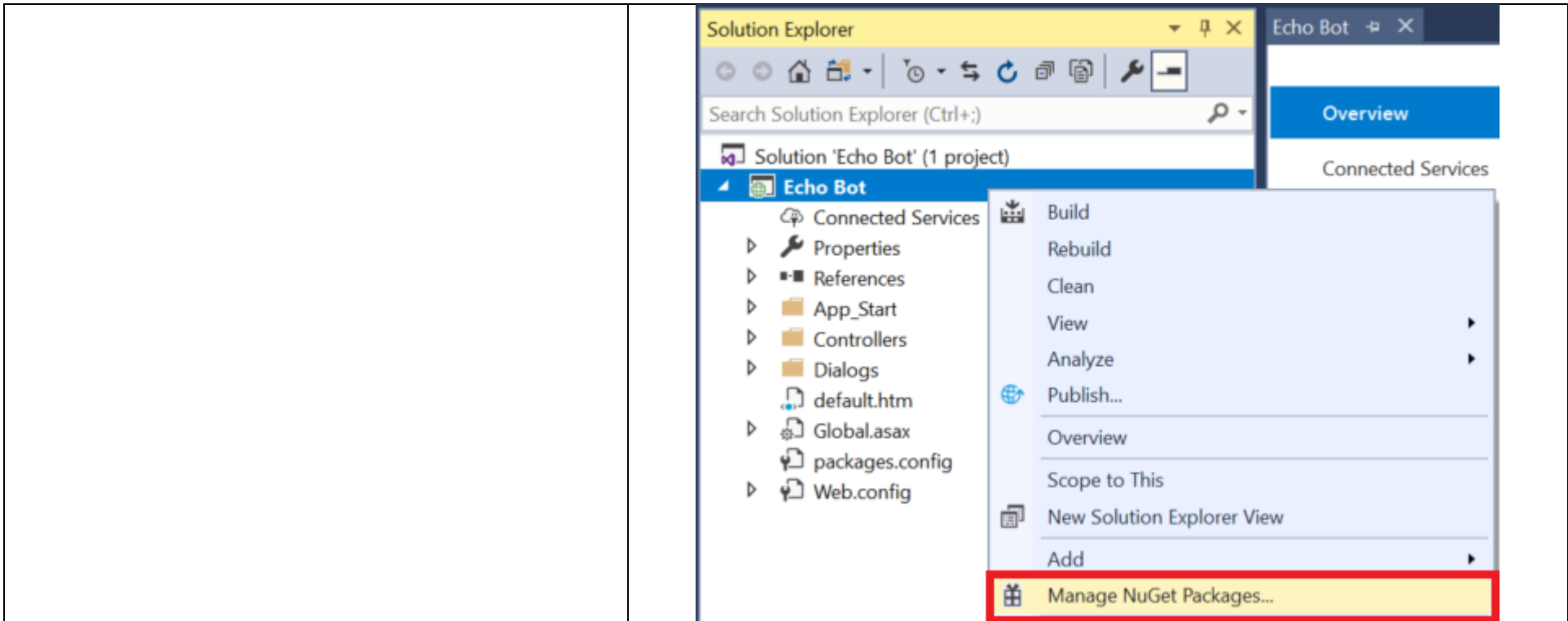
- Creating a Bot using the Visual Studio "Bot Application" template
- Testing a Bot using the Bot Framework Channel Emulator

Steps	Screenshots / Notes
<ol style="list-style-type: none"> Download and install Visual Studio for Windows (if you haven't done so already). Update all of the extensions to their latest versions. Download: <ul style="list-style-type: none"> Bot Application.zip, Bot Controller.zip, and Bot Dialog.zip files <p>Install the project and item templates by copying the zipped files to the C# item project templates directory.</p> <p>NOTE: If you have the Visual Studio workloads installed but don't see the Visual C# folder, your VS templates paths need to be rebuilt. To do this, open the Developer Command Prompt for visual studio and use this syntax:</p> <pre>devenv.exe /InstallVSTemplates</pre> Next, open Visual Studio and create a new C# project. Choose the Bot Application template for your new project. Name the bot Echo Bot. <p>NOTE: Visual might say you need to download and install IIS Express. Please install to continue.</p>	<ul style="list-style-type: none"> Visual Studio <ul style="list-style-type: none"> https://www.visualstudio.com/downloads/ Install these workloads:  Bot Application in ProjectTemplates; Controller & Dialog in ItemTemplates  IIS Express <ul style="list-style-type: none"> https://www.microsoft.com/en-us/download/details.aspx?id=48264 Bot Emulator <ul style="list-style-type: none"> https://emulator.botframework.com/

By using the Bot Application template, you're creating a project that already contains all of the components that are required to build a simple bot, including a reference to the Bot Builder SDK for .NET, Microsoft.Bot.Builder. Verify that your project references the latest version of the SDK:

5. **Right-click** on the project and select **Manage NuGet Packages**.
6. In the **Browse** tab, type "**Microsoft.Bot.Builder**".
7. Locate the **Microsoft.Bot.Builder** package in the list of search results, and click the **Update** button for that package.
8. Follow the prompts to accept the changes and update the package.





First, the Post method within Controllers\MessagesController.cs receives the message from the user and invokes the root dialog.

NOTE: In Lab 1, no code changes are expected. This is simply an example of what you should see in MessagesController.cs in the Controller folder that was created from the Bot Application template.

```
namespace Echo_Bot
{
    [BotAuthentication]
    public class MessagesController : ApiController
    {
        /// <summary>
        /// POST: api/Messages
        /// Receive a message from a user and reply to it
        /// </summary>
        public async Task<HttpResponseMessage> Post([FromBody]Activity
activity)
        {
            if (activity.Type == ActivityTypes.Message)
            {
                await Conversation.SendAsync(activity, () => new
Dialogs.RootDialog());
            }
            else
            {
                HandleSystemMessage(activity);
            }
            var response = Request.CreateResponse(HttpStatusCode.OK);
            return response;
        }
    }
}
```

The root dialog processes the message and generates a response. The `MessageReceivedAsync` method within `Dialogs\RootDialog.cs` sends a reply that echoes back the user's message, prefixed with the text 'You sent' and ending in the text 'which was ## characters', where ## represents the number of characters in the user's message.

NOTE: In Lab 1, no code changes are expected. This is simply an example of what you should see in `RootDialog.cs` in the `Dialogs` folder that was created from the Bot Application template.

```
namespace Echo_Bot.Dialogs
{
    [Serializable]
    public class RootDialog : IDialog<object>
    {
        public Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);

            return Task.CompletedTask;
        }

        private async Task MessageReceivedAsync(IDialogContext context,
            IAwaitable<object> result)
        {
            var activity = await result as Activity;

            // calculate something for us to return
            int length = (activity.Text ?? string.Empty).Length;

            // return our reply to the user
            await context.PostAsync($"You sent {activity.Text} which was
{length} characters");

            context.Wait(MessageReceivedAsync);
        }
    }
}
```

Test your bot

- Next, test your bot by using the Bot Framework Emulator to see it in action. The emulator is a desktop application that lets you test and debug your bot on localhost or running remotely through a tunnel.

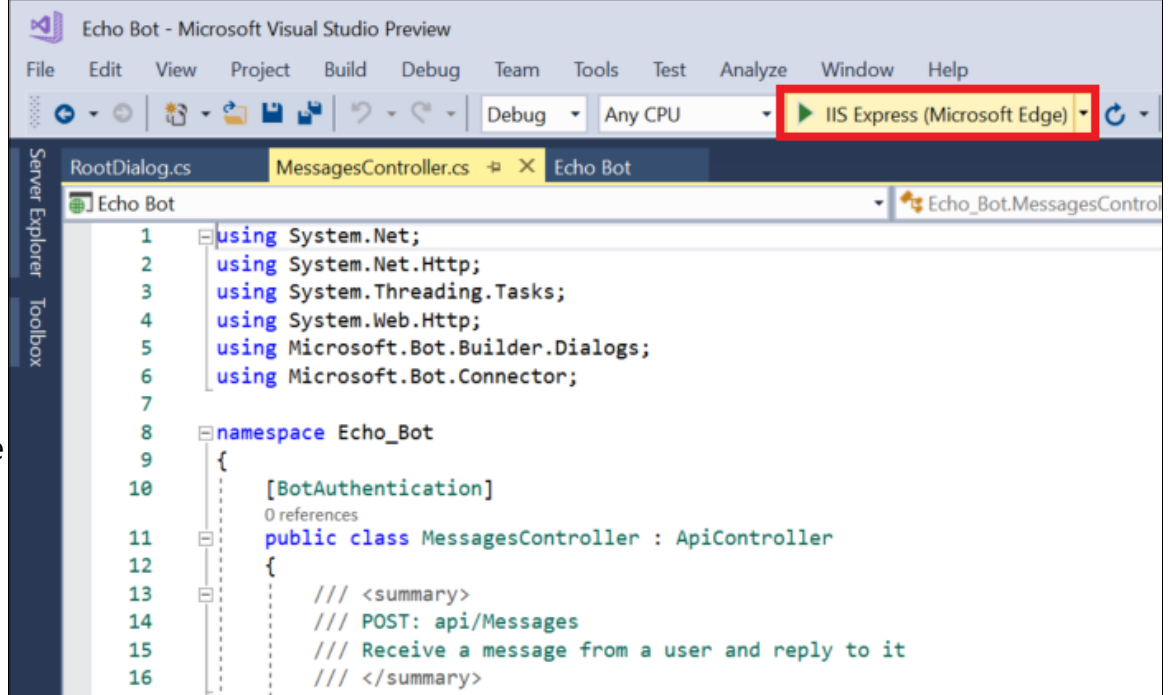
NOTE: First, you'll need to download and install the emulator. After the download completes, launch the executable and complete the installation process.

- Bot Emulator (this link is also in the Prerequisites folder of the lab materials)
 - <https://emulator.botframework.com/>

Start your bot

- After installing the emulator, start your bot in Visual Studio by running IIS Express using a browser as the application host and the Bot Framework Emulator. This Visual Studio screenshot shows that the bot will launch in Microsoft Edge when the run button is clicked.

NOTE: You can change the default browser by clicking the dropdown carrot to the right of the button.



When you click the run button, Visual Studio will build the application, deploy it to localhost, and launch the web browser to display the application's **default.htm** page. For example, here's the application's **default.htm** page shown in Microsoft Edge:

NOTE: You can modify the **default.htm** file within your project to specify the name and description of your bot application. This does not change the behavior of your bot.

Start the emulator and connect your bot

****At the time of this writing Bot Emulator V4 PREVIEW is in pre-release. Instructions could vary and the screenshots may not represent the final release product.****

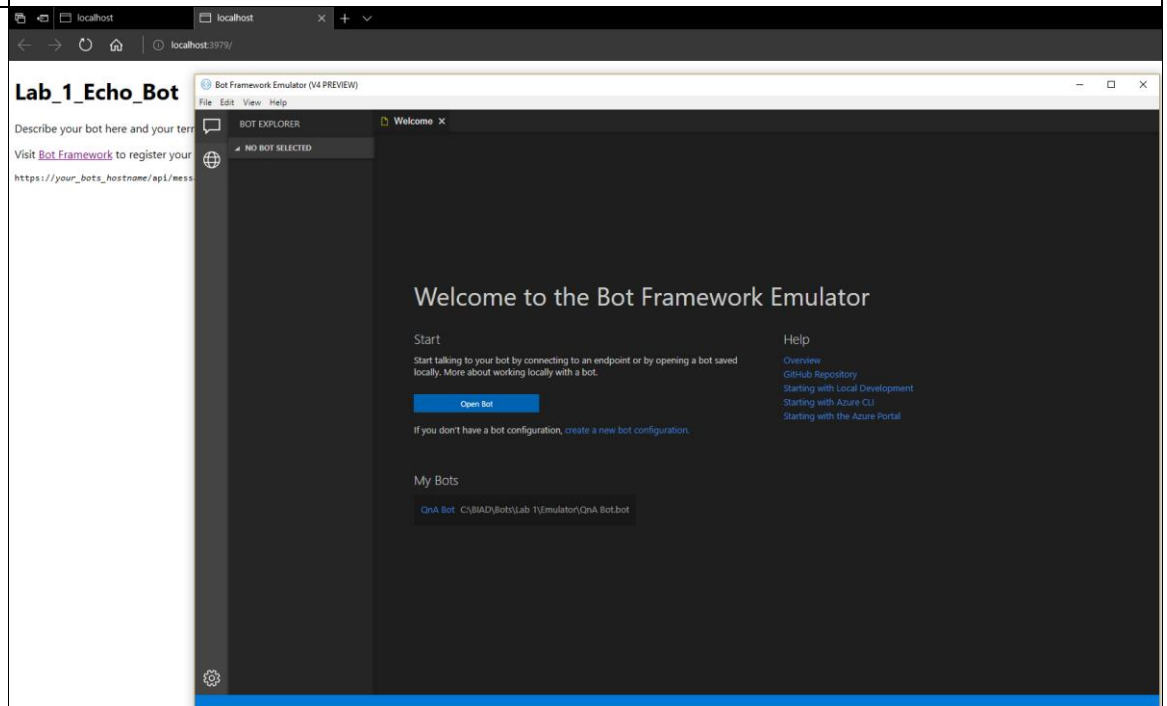
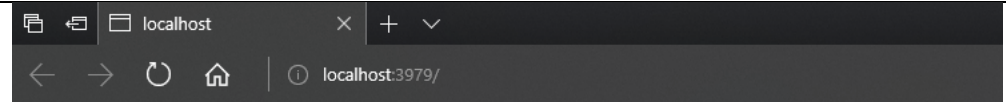
At this point, your bot is running locally.

11. Next, start the emulator and select **create a new bot configuration**.
12. Name your bot something you will remember, e.g. : Echo Bot
13. Then type the endpoint URL:

`http://localhost:port-number/api/messages`

into the text box, where **port-number** matches the port number shown in the browser where your application is running.

HINT: The default port is 3979.



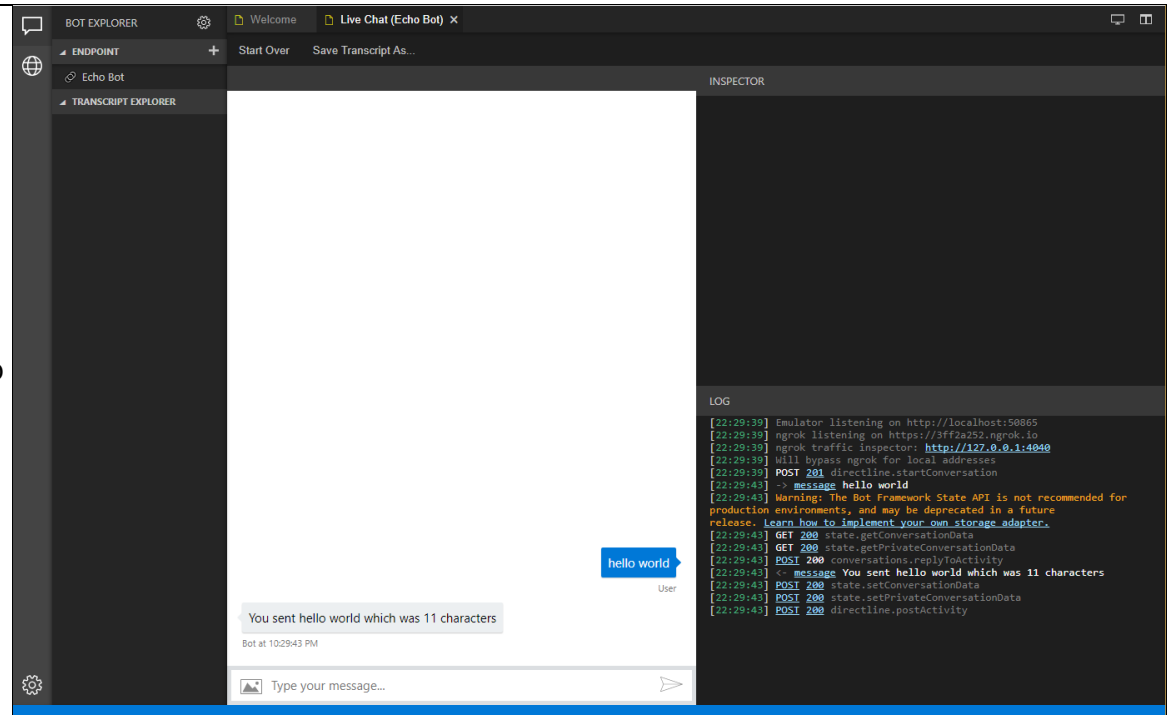
14. Click **Save and Connect**. You won't need to specify **Microsoft App ID** and **Microsoft App Password**. You can leave these fields blank for now. You'll get this information later if you decide to [register your bot](#) in Azure Bot Service.

NOTE: In the screenshot shown, the application is running on port number 3979, so the emulator address would be set to: <http://localhost:3979/api/messages>

Be sure to drop the “s” from https://...

Be sure to explore the new bot emulator and all it has to offer. As a bot developer, this will be the principal tool we will use for the workshop and for planning, testing, and our refining our bots. The new emulator includes a presentation mode so you can demonstrate your bot in front of stakeholders with a clean UI. It also features a split editor so you can run multiple bots at once. It also has direct support for Azure Bot Service, Dispatch apps, LUIS, and QnA Maker endpoint services including LUIS model training within the emulator!

Congratulations! You built your first bot. You've successfully created a bot by using the Bot Application template using the Bot Builder SDK for .NET!

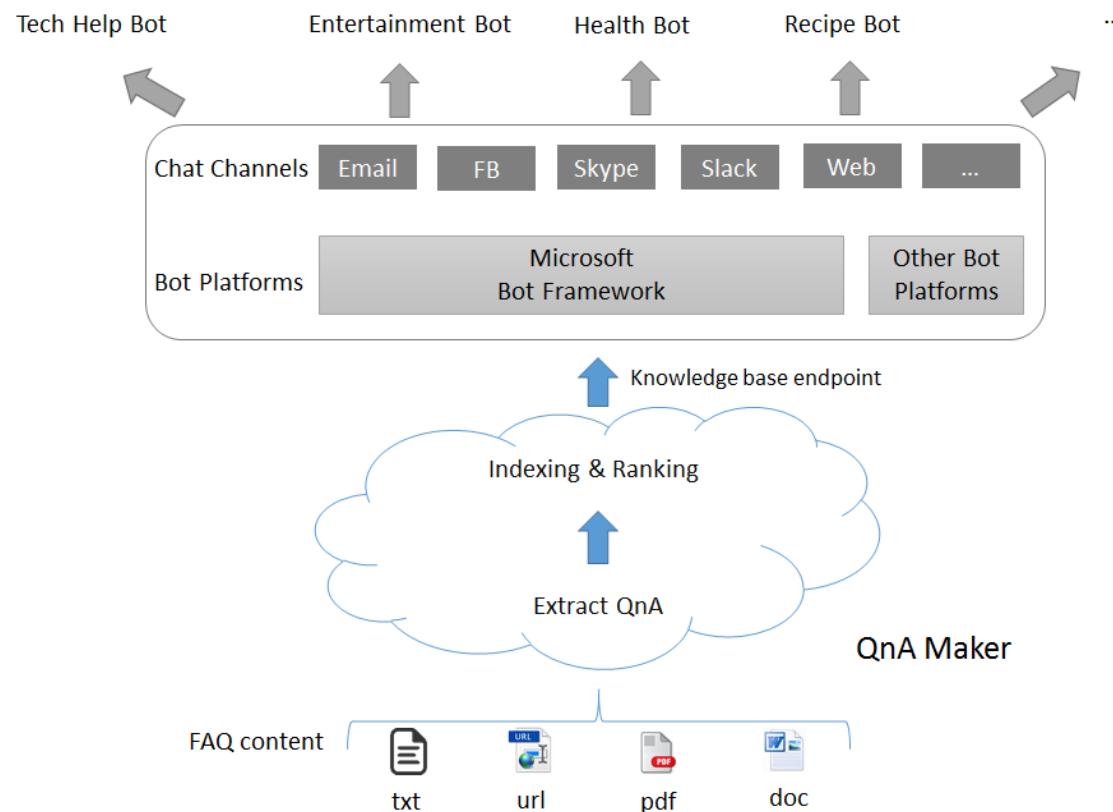


Lab 1.1 – Develop bots by leveraging QnA Maker

One of the basic requirements in writing your own Bot service is to seed it with questions and answers. In many cases, the questions and answers already exist in content like FAQ URLs/documents, etc.

Microsoft QnA Maker is a free, easy-to-use, REST API and web-based service that trains AI to respond to user's questions in a more natural, conversational way. Compatible across development platforms, hosting services, and channels, QnA Maker is the only question and answer service with a graphical user interface—meaning you don't need to be a developer to train, manage, and use it for a wide range of solutions.

With optimized machine learning logic and the ability to integrate industry-leading language processing with ease, QnA Maker distills masses of information into distinct, helpful answers.



Learning Objectives

Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft's Bot Framework.

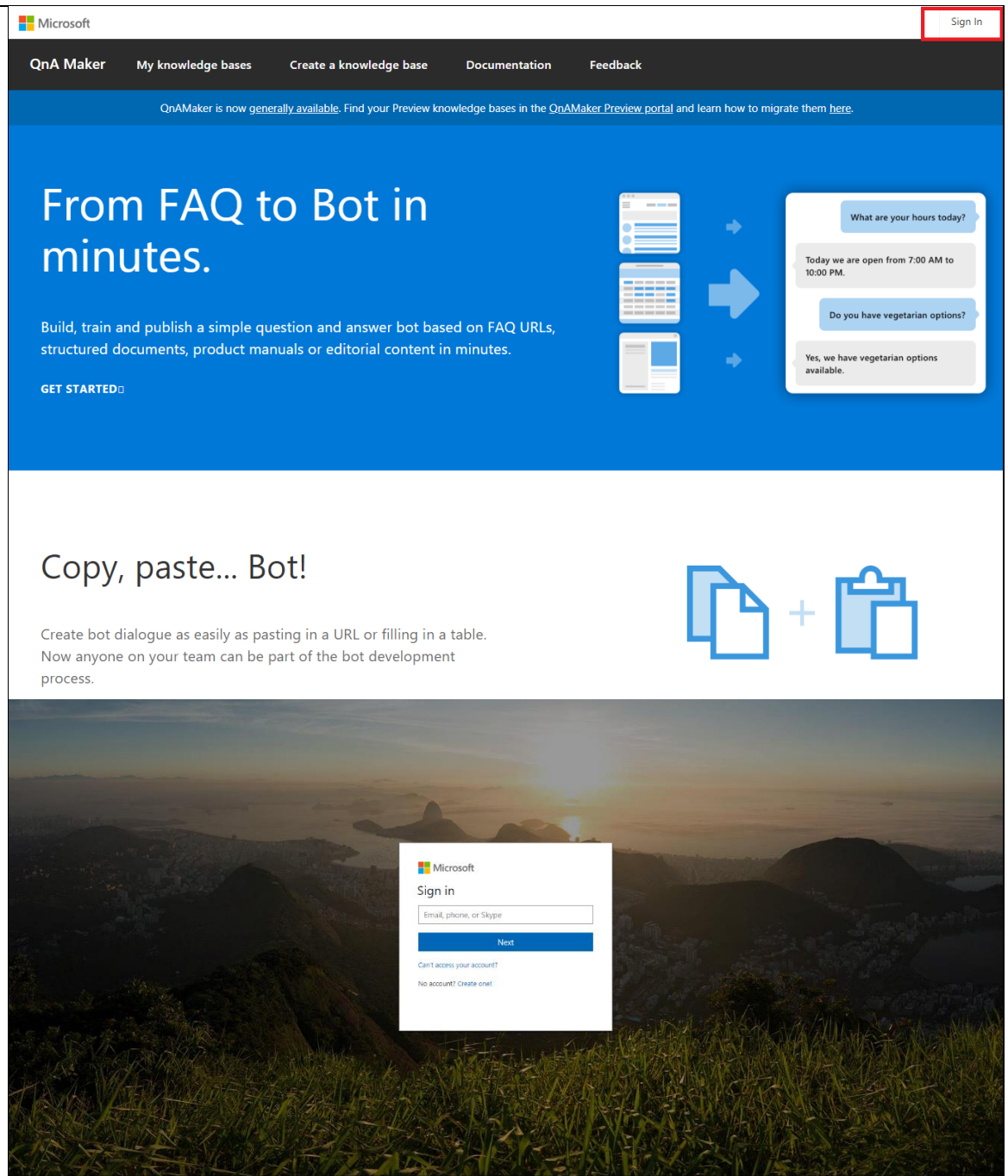
- Creating and Training a Knowledge Base using QnA Maker Service by Microsoft Cognitive Services
- Creating and Connecting the QnA Maker Service to your Bot

Creating and Training our Knowledge Base using QnA Maker Service

Steps	Screenshots / Notes
<p>A good bot is one that is actually useful to users—seems intuitive right? Some bots may need to process payments, handle forms, track location, or perform some other sort of complex or custom operation. We also have Language Understanding Cognitive Service, which can be utilized to provide bots with natural language understanding, so that the proper actions can be called by a user's intent. However, often your bot will simply need to provide well-defined answers to questions that users may have. Visiting the QnA Maker portal, you can discover why it is such a useful tool for creating FAQ knowledge-based question-answer functionality for your bot. In this article, we'll be exploring different ways to utilize the service to help create better bots.</p>	<p>Prerequisites</p> <ul style="list-style-type: none">• Microsoft Account (QnA Service, Bot registration) – Free <p>Navigate to:</p> <ul style="list-style-type: none">• https://qnamaker.ai

First, we will need to sign in to qnamaker.ai using a Microsoft account. If you have a Microsoft Azure subscription, you will want to use it. If you do not have a Microsoft account, you can sign up for one—it's free!

[Microsoft Account Sign Up](#)



The screenshot displays the Microsoft QnA Maker website. At the top, the Microsoft logo is on the left, and a 'Sign in' button is on the right. The navigation bar includes 'QnA Maker', 'My knowledge bases', 'Create a knowledge base', 'Documentation', and 'Feedback'. A banner below the navigation bar states: 'QnAMaker is now generally available. Find your Preview knowledge bases in the QnAMaker Preview portal and learn how to migrate them here.' The main content area has a blue background with the heading 'From FAQ to Bot in minutes.' Below this, it says: 'Build, train and publish a simple question and answer bot based on FAQ URLs, structured documents, product manuals or editorial content in minutes.' A 'GET STARTED' button is present. To the right, a diagram shows a flow from input sources (FAQs, documents, manuals) to a chatbot interface. The chatbot interface shows a conversation: 'What are your hours today?' followed by 'Today we are open from 7:00 AM to 10:00 PM.' and 'Do you have vegetarian options?' followed by 'Yes, we have vegetarian options available.' Below this, the text 'Copy, paste... Bot!' is displayed, followed by 'Create bot dialogue as easily as pasting in a URL or filling in a table. Now anyone on your team can be part of the bot development process.' To the right of this text is an icon representing a document being added to a clipboard. The bottom of the screenshot shows a sign-in dialog box over a scenic background. The dialog box has the Microsoft logo, the text 'Sign in', a text input field for 'Email, phone, or Skype', a 'Next' button, and links for 'Can't access your account?' and 'No account? Create one!'.

Next, we will create a new QnA service. To do this, you will select **Create a knowledge base**.

Then, you will select **Create a QnA service**. This will require an Azure subscription.

[Click here to sign up for an Azure subscription.](#)

This workshop assumes you already have a subscription. If not, do not worry, it is free to sign up. If you are a first-time user, Microsoft provides you with \$200 in Azure credits to use towards your development. But, in order to provision the service it will require a credit card to be associated with your account. This is just to verify you are human. You will not be charged. The free QnA Maker Tier is free to use.

Microsoft

QnA Maker My knowledge bases **Create a knowledge base** Documentation Feedback

QnAMaker is now [generally available](#). Find your Preview knowledge bases in the [QnAMaker Preview portal](#) and learn how to migrate them [here](#).

Create a knowledge base

Create an Azure service for your QnA knowledge base and add sources that contain the question and answer pairs you would like to include.
[Learn more about creating a knowledge base.](#)

STEP 1

Create a QnA service in Microsoft Azure.
Create an Azure QnA service for your KB. If you already have an Azure QnA service for this KB, skip this step.
[Learn more about Azure subscriptions, pricing tiers, and keys.](#)

Create a QnA service

STEP 2

Connect your QnA service to your KB.
After you create an Azure QnA service, [refresh this page](#) and then select your Azure service using the options below.

* Microsoft Azure Directory ID
Select tenant ▼

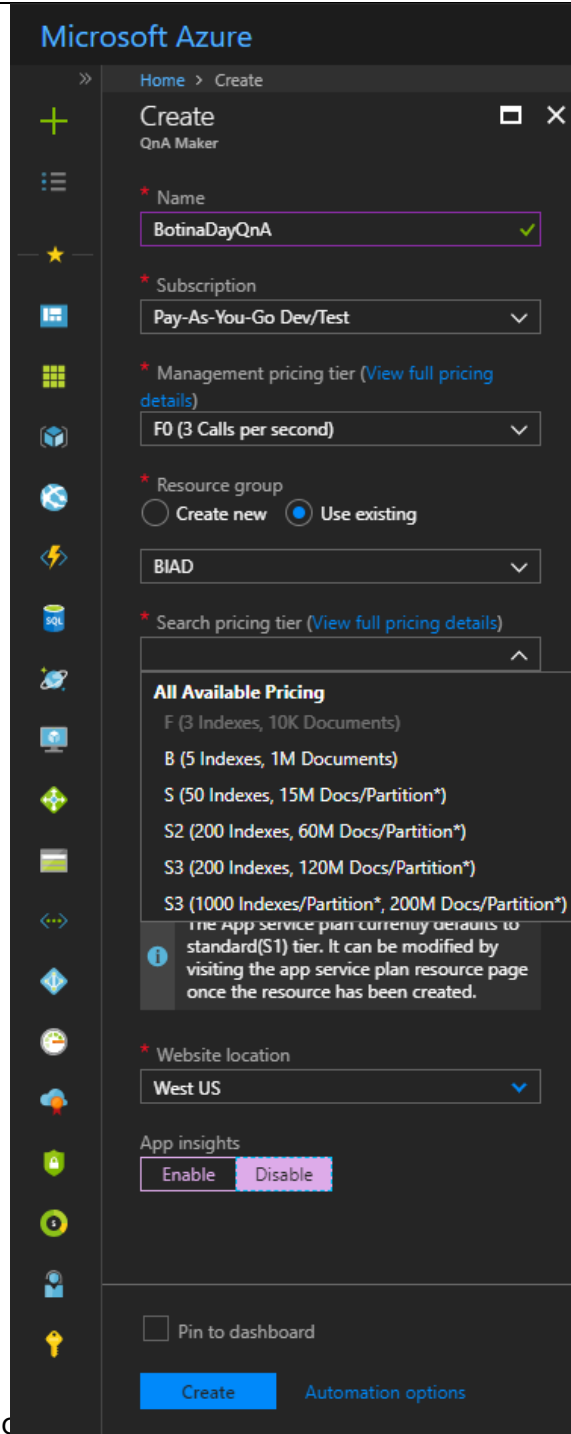
* Azure subscription name
Select subscription ▼

* Azure QnA service
Select service ▼

The button should open the QnA Maker Service Blade, if not click [here](#) to find the service.

Enter in the details by creating a unique name, selecting your subscription and selecting the F0 Management Pricing Tier as well as the F Search Pricing Tier. If you do not have a Resource group created, you can create one now or select an existing one you'd like to use. Once you are satisfied and met all the requirements, select **Create** at the bottom of the blade. The Azure App Service will provision in a couple of minutes.

One new benefit to the general availability of QnA Maker is the ability to store chatlogs via Application Insights to the right → You can also run analytics to gain insights into usage. However, please disable App insights as this does charge you for usage. Read the documentation [here](#) for pricing details.



The screenshot shows the 'Create QnA Maker' blade in the Microsoft Azure portal. The blade is titled 'Create QnA Maker' and has a 'Home > Create' breadcrumb. The left sidebar contains various Azure service icons. The main content area has the following fields and options:

- Name:** 'BotinaDayQnA' (with a green checkmark icon).
- Subscription:** 'Pay-As-You-Go Dev/Test' (dropdown menu).
- Management pricing tier:** 'F0 (3 Calls per second)' (dropdown menu, with a link to 'View full pricing details').
- Resource group:** 'BIAD' (dropdown menu). Radio buttons for 'Create new' and 'Use existing' are present, with 'Use existing' selected.
- Search pricing tier:** A dropdown menu with an upward arrow icon. Below it, a section titled 'All Available Pricing' lists several tiers: 'F (3 Indexes, 10K Documents)', 'B (5 Indexes, 1M Documents)', 'S (50 Indexes, 15M Docs/Partition*)', 'S2 (200 Indexes, 60M Docs/Partition*)', 'S3 (200 Indexes, 120M Docs/Partition*)', and 'S3 (1000 Indexes/Partition*, 200M Docs/Partition*)'. A note below the tiers states: 'The app service plan currently defaults to standard(S1) tier. It can be modified by visiting the app service plan resource page once the resource has been created.'
- Website location:** 'West US' (dropdown menu).
- App insights:** Two buttons, 'Enable' and 'Disable', with 'Disable' highlighted in blue.
- Pin to dashboard:** A checkbox that is currently unchecked.
- Buttons:** A blue 'Create' button and a blue 'Automation options' link.

Once your provisioned services have completed successfully, back in qnamaker.ai, you will connect QnA Maker service to this knowledge base.

Name your KB.

Populate your KB.

NOTE: You do not have to add a URL or add a file; however, if you are running behind, you can use the "QnA Lab 1_1.tsv" file located in: BIAD > Bots > Lab 1 > QnA Maker Knowledgebase

Then, create your KB.

STEP 3

Name your KB.

The knowledge base name is for your reference and you can change it at anytime.

* Name

BotinaDatLab1

STEP 4

Populate your KB.

Extract question-and-answer pairs from an online FAQ, product manuals, or other files. Supported formats are .tsv, .pdf, .doc, .docx, .xlsx, containing questions and answers in sequence. [Learn more about knowledge base sources](#). Skip this step to add questions and answers manually after creation. The number of sources and file size you can add depends on the QnA service SKU you choose. [Learn more about QnA Maker SKUs](#).

URL

http://

+ Add URL

File name

+ Add file

STEP 5

Create your KB

The tool will look through your documents and create a knowledge base for your service. If you are not using an existing document, the tool will create an empty knowledge base table which you can edit.

Create your KB

In your service's knowledge base editor, you can add question and answer pairs. Think of common things user's will utter, like:

"hi"

"help"

"I'm lost"

"cancel"

"what can you do"

"go back"

Feel free to add a KB document from your organization.

...and when you are satisfied, click on **Save and train**.

Behind the scenes, QnA maker leverages LUIS and other cognitive services to create a model for your service.

To test your newly created service, click **Test** and try a couple of question and answer pairs. What happens when you ask a question that is not in the KB?

If you think of another alternative phrasing, you can enter it here.

If the answer isn't quite right, then you can enter a new answer to the right, but do not

The image displays three screenshots of the BotinaDatLab1 QnA maker interface, illustrating the process of adding and testing question and answer pairs.

Top Screenshot: Knowledge base editor (Initial state)
The interface shows the "Knowledge base" section with a search bar and a list of question and answer pairs. The "Add QnA pair" button is highlighted with a red box. The "Question" field is empty, and the "Answer" field is also empty. The "Save and train" button is highlighted with a red box.

Middle Screenshot: Knowledge base editor (With pairs)
The interface shows the "Knowledge base" section with a search bar and a list of question and answer pairs. The "Add QnA pair" button is highlighted with a red box. The "Question" field contains the text "help", "cancel", "go back", "quit", and "+". The "Answer" field contains the text "I can see you are lost. Just type 'trivia' to play trivia or type 'jokes' to hear a joke." The "Save and train" button is highlighted with a red box.

Bottom Screenshot: Test/Inspect view
The interface shows the "Test" and "Inspect" views. The "Test" view displays a chat history with the following messages:
- User: "help"
- Bot: "I can see you are lost. Just type 'trivia' to play trivia or type 'jokes' to hear a joke."
- User: "hi"
- Bot: "Hello there, I am BIAD bot."
The "Inspect" view shows the "Question" field with the text "help" and the "Answer" field with the text "I can see you are lost. Just type 'trivia' to play trivia or type 'jokes' to hear a joke." The "Confidence score" is 100. The "Add alternative phrasing" field is empty.

forget to **Save and train** your model for QnA Maker service to reflect these changes. Once your satisfied with the changes, go ahead and select **Publish**.

Then, **Publish** once more to finalize the changes.

NOTE: Do not worry, you can always come back and modify your knowledgebase. Just don't forget to **Save and train** every time you do.

After your service is deployed, you'll be re-directed to a page view which provides a sample HTTP request for another application to use the service. Also provided are the **Knowledgebase Id**, **Host URL** and the **Endpoint Key** to access the service. Copy and paste these unique values and save them for later, as we'll need to pass these to the bot later.

That's it for creating and deploying our QnA service. You can always go back to add more question-answer pairs and retrain and publish your QnA knowledge base at any time.

BotinaDatLab1

EDIT

PUBLISH

BotinaDatLab1

Your service has never been deployed.

Publishing your knowledge base moves your QnAs from the test index to the production index. Once you publish, the knowledge base endpoint becomes available for use in your Bot or App

This knowledge base will be published to the [biadqnamaker](#) QnAMaker service.

Cancel

Publish

Success! Your service has been deployed. What's next?

You can always find the deployment details in your service's settings.

Use the below HTTP request to build your bot. [Learn how.](#)

Sample HTTP request

POST /knowledgebases/ Knowledge base ID /generateAnswer

Host: QnA Host URL

Authorization: EndpointKey QnA Endpoint Key

Content-Type: application/json

{"question": "<Your question>"}

Need to fine-tune and refine? Go back and keep editing your service.

Edit Service

Version 5.18

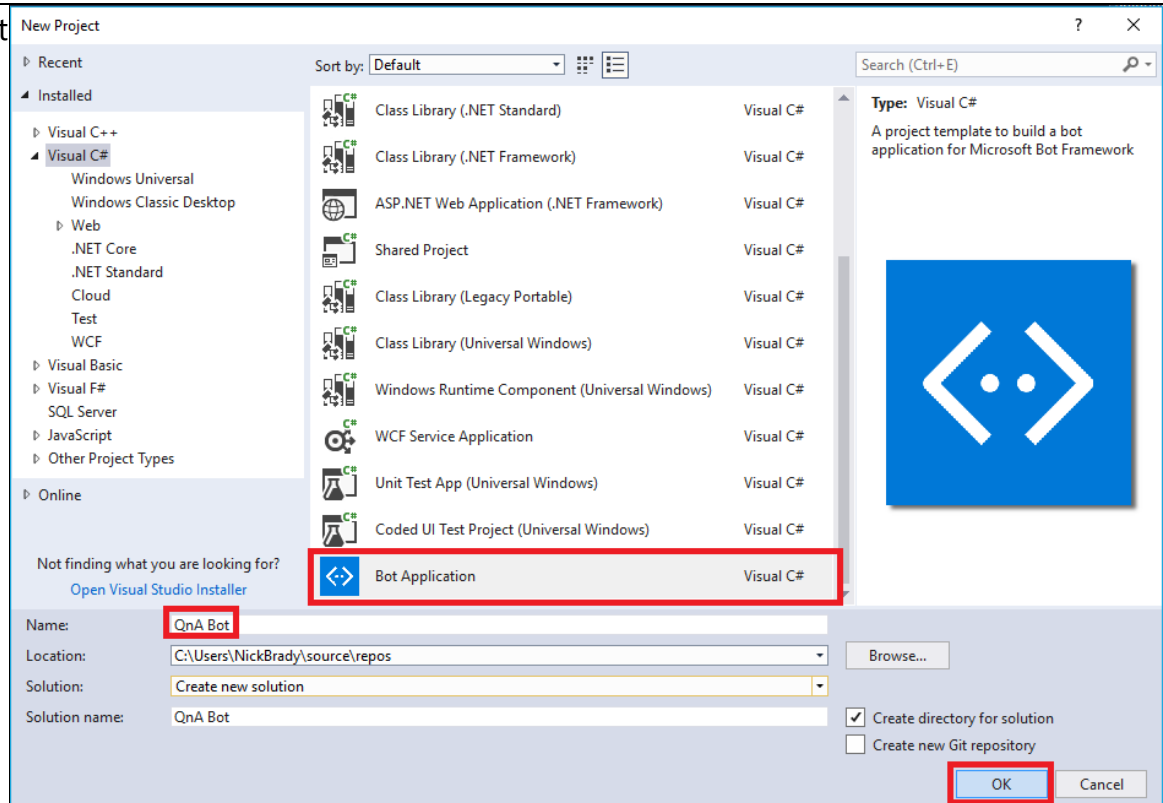
© 2018 Microsoft

25 | Page

Creating and Connecting a Bot to QnA Maker Service

In this step we'll create a new bot using the Bot Application template again. You can do this by opening Visual Studio, then selecting 'File -> New Project -> Bot Application' as shown to the right.

NOTE: Ensure you have the Bot Application template in the ProjectTemplates > Visual C# folder.



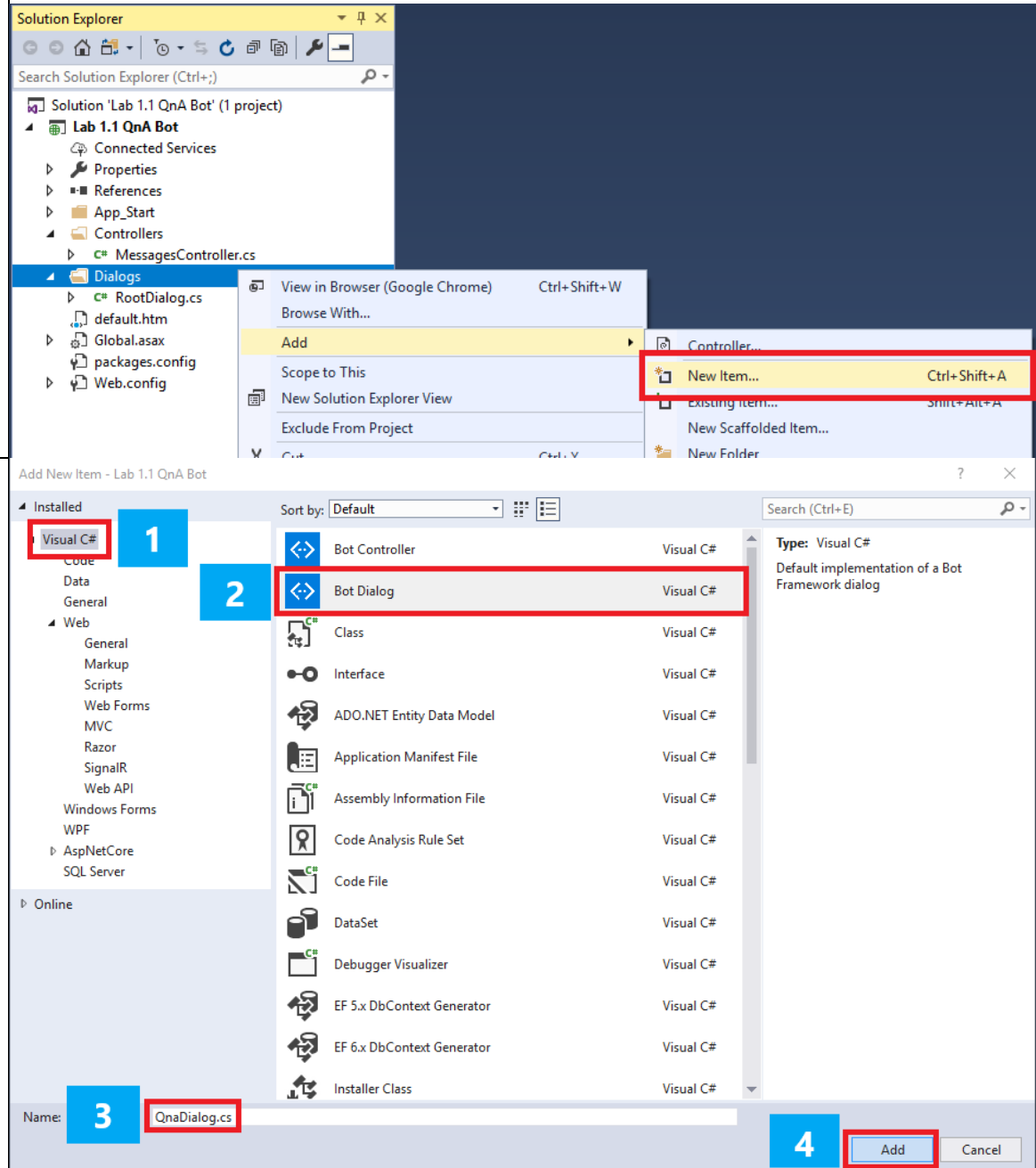
Within the Solution Explorer in Visual Studio, in your project's Web.config, in the <configuration> settings add the following keys – **QnAHost**, **QnAEndPointKey** and **QnaKnowledgebaseId**. The values for these keys are provided by your QnA service upon successful deployment (see page 25) which we saved earlier when we created QnA service.

```
<appSettings>
  <!--<add key="BotId" value="YourBotId" />-->
  <!--<add key="MicrosoftAppId" value="" />-->
  <!--<add key="MicrosoftAppPassword" value="" />-->
  <add key="QnAHost" value="ENTER_HOST_URL_HERE" />
  <add key="QnAEndPointKey" value="ENTER_ENDPOINT_KEY_HERE" />
  <add key="QnAKnowledgebaseId" value="ENTER_KNOWLEDGE_BASE_ID_HERE" />
</appSettings>
```

Next, we'll create a new Dialog for our bot – **QnaDialog.cs**

To do this, let's start by right-clicking on the Dialogs folder within the solution to add a **New Item...**

1. Make sure Visual C# is highlighted in the Add New Item Installed list.
2. Select Bot Dialog
3. Change the name to QnaDialog.cs
4. Select Add



You should receive some boilerplate code shown to the right that is ready for us to add logic.

NOTE: We will be replacing this code in the next step.

```
using System;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace ENTER_SOLUTION_NAME_HERE.Dialogs
{
    [Serializable]
    public class QnaDialog : IDialog<object>
    {
        public Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);

            return Task.CompletedTask;
        }

        private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
        {
            var activity = await result as IMessageActivity;

            // TODO: Put logic for handling user message here

            context.Wait(MessageReceivedAsync);
        }
    }
}
```

Replace the **QnaDialog.cs** code with the code to the right.

NOTE: Use the Code Snippets provided in the Bot > Lab 1 > Code Snippets > Lab 1_1 folder.

```
using System;
using System.Configuration;
using Microsoft.Bot.Builder.Dialogs;
using System.Threading.Tasks;
using Microsoft.Bot.Connector;
using System.Web;

namespace ENTER_SOLUTION_NAME_HERE.Dialogs
{
    [Serializable]
    public class QnaDialog : IDialog<object>
    {
        private QnAMakerService _QnAService = new QnAMakerService(
            ConfigurationManager.AppSettings["QnaHost"],
            ConfigurationManager.AppSettings["QnaKnowledgebaseId"],
            ConfigurationManager.AppSettings["QnaEndPointKey"]);

        public Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
            return Task.CompletedTask;
        }

        public async Task MessageReceivedAsync(IDialogContext context,
            IAwaitable<IMessageActivity> result)
        {
            string safeText =
                HttpUtility.UrlEncode(((IMessageActivity)context.Activity).Text);
            string answer = _QnAService.GetAnswer(safeText);
            if (string.IsNullOrEmpty(answer))
            {
                await context.PostAsync("No good match found in KB.");
            }
            else
            {
                await context.PostAsync(answer);
            }
        }
    }
}
```

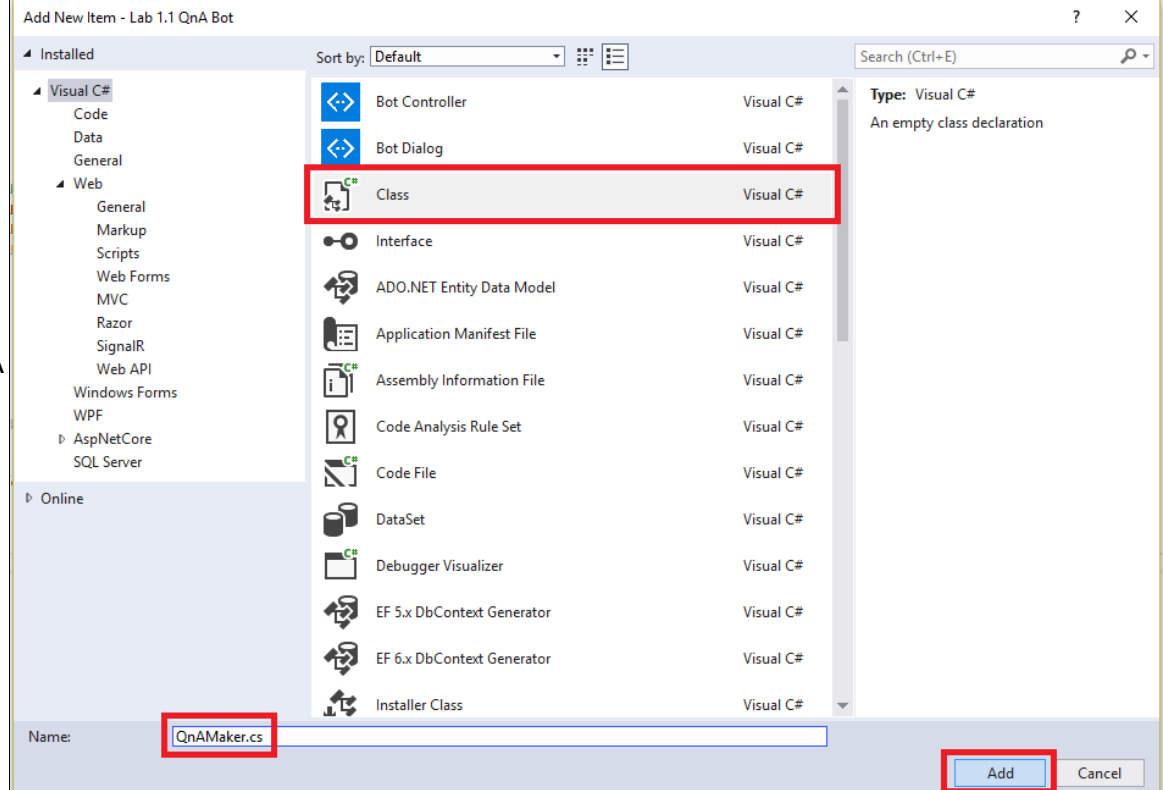
Next, we will add a new C# class so our QnaDialog can use the QnAMaker service.

Let's add another **New Item...** to the 'Dialogs' folder.

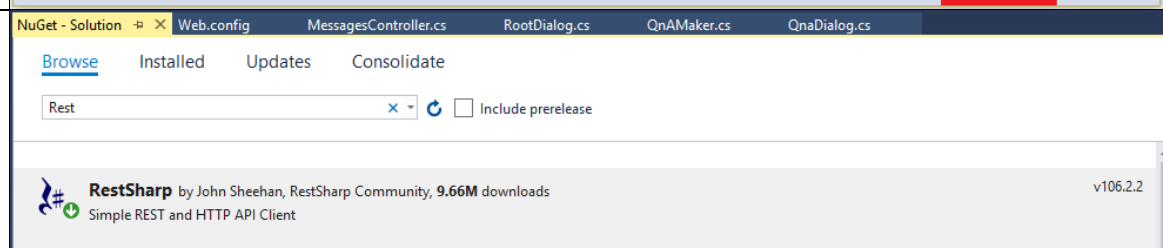
This time, we will select the C# Class.

Give it an appropriate title to tie it back to QnA Maker.

Then, select **Add**.



This code will require the RestSharp NuGet package. Right-click on your solution to open the context menu or in Menu > Tools to manage and install this NuGet package. *Don't forget to update all packages too!*



Next, enter the code to the right →

NOTE: A code snippet has been provided for you in the workshop Bots > Lab 1 > Code Snippets > Lab 1_1 folder.

```
using Newtonsoft.Json;
using RestSharp;
using System;
using System.Collections.Generic;

namespace Lab_1_1_QnA_Bot.Dialogs
{
    /// <summary>
    /// QnAMakerService is a wrapper over the QnA Maker REST endpoint
    /// </summary>
    [Serializable]
    public class QnAMakerService
    {
        private string qnaServiceHostName;
        private string knowledgeBaseId;
        private string endpointKey;

        /// <summary>
        /// Initialize a particular endpoint with its details
        /// </summary>
        /// <param name="hostName">Hostname of the endpoint</param>
        /// <param name="kbId">Knowledge base ID</param>
        /// <param name="ek">Endpoint Key</param>
        public QnAMakerService(string hostName, string kbId, string ek)
        {
            qnaServiceHostName = hostName;
            knowledgeBaseId = kbId;
            endpointKey = ek;
        }

        /// <summary>
        /// Call the QnA Maker endpoint and get a response
        /// </summary>
        /// <param name="query">User question</param>
        /// <returns></returns>
        public string GetAnswer(string query)
```

```

        {
            var client = new RestClient(qnaServiceHostName +
                "/knowledgebases/" + knowledgeBaseId + "/generateAnswer");
            var request = new RestRequest(Method.POST);
            request.AddHeader("authorization", "EndpointKey " +
                endpointKey);
            request.AddHeader("content-type", "application/json");
            request.AddParameter("application/json", "{\"question\": \"" +
                query + "\"}", ParameterType.RequestBody);
            IRestResponse response = client.Execute(request);

            if (response.StatusCode == System.Net.HttpStatusCode.OK)
            {
                // Deserialize the response JSON
                QnAAnswer answer =
                JsonConvert.DeserializeObject<QnAAnswer>(response.Content);

                // Return the answer if present
                if (answer.Answers.Count > 0)
                {
                    if (answer.Answers[0].answer != "No good match found
                    in KB.")
                    {
                        return answer.Answers[0].answer;
                    }
                    else
                    {
                        return "";
                    }
                }
                else
                {
                    return "";
                }
            }
            else

```



```

        {
            throw new Exception($"QnAMaker call failed with status
code {response.StatusCode} {response.StatusDescription}");
        }
    }

    }

    /* START - QnA Maker Response Class */
    public class Metadata
    {
        public string Name { get; set; }
        public string Value { get; set; }
    }

    public class Answer
    {
        public IList<string> Questions { get; set; }
        public string answer { get; set; }
        public double Score { get; set; }
        public int Id { get; set; }
        public string Source { get; set; }
        public IList<object> Keywords { get; set; }
        public IList<Metadata> Metadata { get; set; }
    }

    public class QnAAnswer
    {
        public IList<Answer> Answers { get; set; }
    }

    /* END - QnA Maker Response Class */
}

```

Next, go into **MessagesController.cs** and edit the Post method as shown:

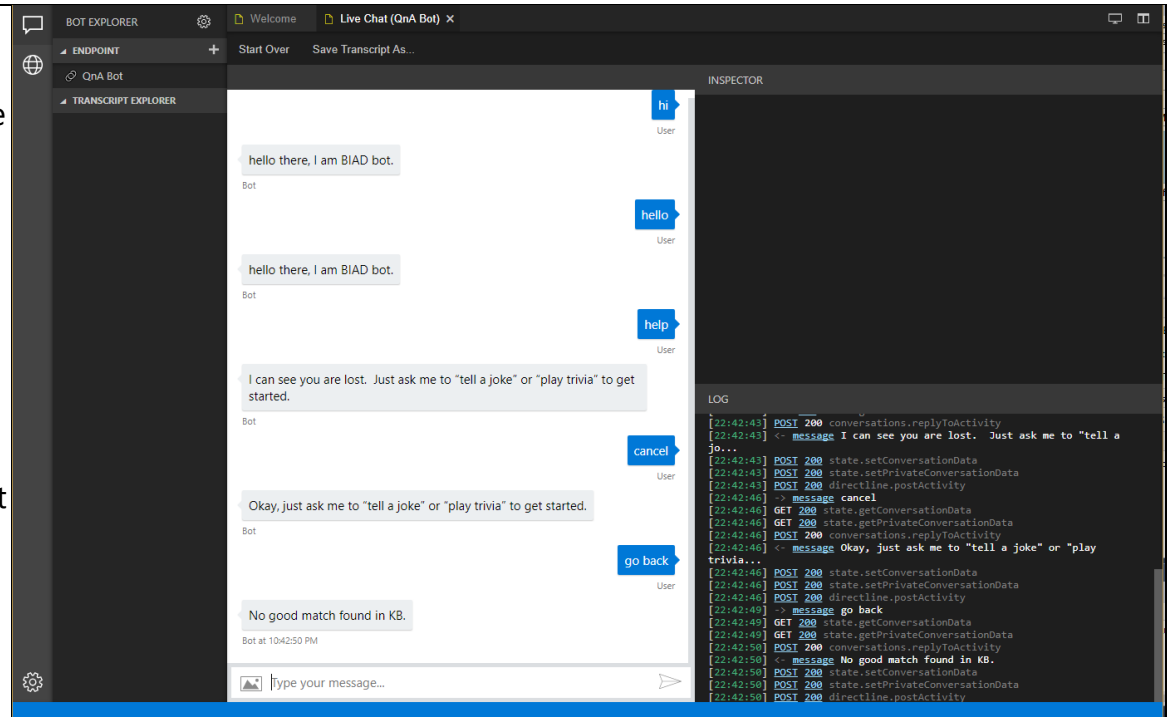
The only thing done in the code snippet to the right is we commented out the default call to the RootDialog (provided by the Bot Application Template), and we're passing all incoming message activity to the QnaDialog.

NOTE: It is up to the developer to determine how their bot dialogs and conversation flow are managed, this is simply a quick way for our bot demo to call the QnA service but it is not necessarily how you should manage this dialog in production.

At this point, everything we need to run the bot and interact with our QnA service is ready. Launching our bot and running the emulator locally, we can interact with the bot to verify that our QnA service is working properly.

```
public class MessagesController : ApiController
{
    /// <summary>
    /// POST: api/Messages
    /// Receive a message from a user and reply to it
    /// </summary>
    public async Task<HttpResponseMessage> Post([FromBody]Activity
activity)
    {
        if (activity.Type == ActivityTypes.Message)
        {
            await Conversation.SendAsync(activity, () => new
Dialogs.QnaDialog());
            // await Conversation.SendAsync(activity, () => new
Dialogs.RootDialog());
        }
        else
        {
            HandleSystemMessage(activity);
        }
        var response = Request.CreateResponse(HttpStatusCode.OK);
        return response;
    }
}
```

So far, so good! Since our demo bot is only using one dialog (QnaDialog), we can be certain that response activity that we see in the emulator is coming from the QnA service. The service also provides natural language processing behind the scenes. You'll even notice that basic markdown formatting and hyperlinks are supported for web chat, although this may vary depending on the channel your bot is connected to. While this is great, having simple text and markdown formatting may not be enough. As modern bot developers, we also want to provide a good experience for our users.



Lab 1.2 – QnA Maker with rich cards in .NET

It is quite common for bots to interpret natural language. But natural language processing (NLP) can get quite tricky and extracting semantic representations in text is still an open research. A solution around avoiding large use of text is to leverage a rich User Interface (UI).

For example, you could ask a user “what kind of pizza would you like?”. This would result in a near infinite number of possible pizza varieties if you include all the combinations via free text. A more finite way of asking a user “what kind of pizza would you like?” is to present a list of options from which the user can select. This would avoid working with free text making the task of processing much easier for the Bot.

The aim of this lab is to demonstrate integration of user controls such as prompts, buttons and menus with bots.

For this next lab,

Documentation:

- Rich Cards - <https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-add-rich-card-attachments>

Learning Objectives

Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft’s Bot Framework.

- Creating rich cards using .NET
- Working with simple markdown in QnA Maker to render rich card experiences

Adding delimiters to QnA and Rich Card Creation

Before adding any more code, we need to revisit our QnA service. The initial question answer pairs where we entered into the knowledge base were very direct – we entered a question, and provided the exact answer that we wanted a user to see from the service.

Since our goal is to format the response into a more visually appealing UI card for users, we need to re-think the way we are storing answers. An answer we store can be a simple text string, but a UI card is often composed of more elements of data. There could be a title, description, image, links, etc. all of which together provide the data we need to create a card attachment.

So, what we can do is use our knowledge base to not just store an answer, but we can also use the QnA service to store data. Below, we'll train and re-publish our knowledge base with one more question-answer pair, only this time we'll be formatting our answers to store more data.

Question	Answer
Original source: QnA Lab 1,2.tsv	
Question X +	Answer
hi X hello X +	hello there, I am BIAD bot.
what can you do X +	I can tell jokes and play trivia. Just ask me to "tell a joke" or "play trivia" to get started.
help X +	I can see you are lost. Just ask me to "tell a joke" or "play trivia" to get started.
cancel X +	Okay, just ask me to "tell a joke" or "play trivia" to get started.
What is Bot Framework? X +	Bot Framework Bot Framework is a set of tools, services, connectors and Bot Builder SDKs. Developers can get started in seconds with out-of-the-box templates for scenarios including basic, form, language understanding, Q&A, and proactive bots https://dev.botframework.com/ https://dev.botframework.com/Client/Images/ChatBot-BotFramework.png

In our new question-answer pair, we are storing a title, description, URL, and image link to be rendered to our card, separated by vertical bars—or the “|” symbol. This is simply one format which is possible, you could also store your data in JSON format. However, **it is recommended that you maintain a consistent format for your data** in the knowledge base, for reasons which will become apparent soon as we develop the code in our demo bot.

NOTE: Here is an example of the .tsv file found in the:
Bots > QnA Maker Knowledgebase →

Question	Answer	Source
hi	hello there, I am BIAD bot.	Editorial
hello	hello there, I am BIAD bot.	Editorial
what can you do	I can tell jokes and play trivia. Just ask me to "tell a joke" or "play trivia" to get started.	Editorial
help	I can see you are lost. Just ask me to "tell a joke" or "play trivia" to get started.	Editorial
cancel	Okay, just ask me to "tell a joke" or "play trivia" to get started.	Editorial
What is Bot Framework?	Bot Framework Bot Framework is a set of tools, services, connectors and Bot Builder SDKs. Developers can get started in seconds with out-of-the-box templates for scenarios including basic, form, language understanding, Q&A, and proactive bots. https://dev.botframework.com/ https://dev.botframework.com/Client/Images/ChatBotBotFramework.png	Editorial



Provided here, is an implementation of a hero card, using the data format we chose in our knowledge base. Replace the original **RootDialog.cs** code with what is provided on the right →

Here's a breakdown of this implementation:

1. The RootDialog receives the activity from the MessagesController.
2. The RootDialog starts an asynchronous process and pass the context to the MessageReceivedAsync method.
3. This method then forwards the context to the QnaDialog to find a match in the KB.
4. After we find the result in QnaDialog, we pass the answer back to the RootDialog in the AfterQnA method.
5. This method builds a messages after meeting some conditions based upon whether an answer was available.
6. Because we defined our answer format previously in our knowledge base, we know that the data we want to capture is stored as string elements separated by a vertical bar ('|'), so we can use a string split to store each property separately.
7. A new HeroCard object called 'card' is created, the card is then formatted using the information from the QnA answer.

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

namespace ENTER_SOLUTION_NAME_HERE.Dialogs
{
    [Serializable]
    public class RootDialog : IDialog<object>
    {
        public Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
            return Task.CompletedTask;
        }

        private async Task MessageReceivedAsync(IDialogContext context,
            IAwaitable<object> result)
        {
            var activity = await result as Activity;
            await context.Forward(new QnaDialog(), AfterQnA, activity,
                CancellationToken.None);
        }

        private async Task AfterQnA(IDialogContext context,
            IAwaitable<object> result)
        {
            string message = null;

            try
            {
                message = (string)await result;
            }
            catch (Exception e)
            {
            }
        }
    }
}
```

8. The card is added to the reply activity as an attachment, before being posted back to the user.
9. Otherwise, we tell the user, "No good match was found in the KB" and echo back what the user said.

```
{
    await context.PostAsync($"QnAMaker: {e.Message}");
    // Wait for the next message
    context.Wait(MessageReceivedAsync);
}

// Display the answer from QnA Maker Service
var answer = message;

if (!string.IsNullOrEmpty(answer))
{
    Activity reply =
((Activity)context.Activity).CreateReply();

    string[] qnaAnswerData = answer.Split('|');
    string title = qnaAnswerData[0];
    string description = qnaAnswerData[1];
    string url = qnaAnswerData[2];
    string imageURL = qnaAnswerData[3];

    HeroCard card = new HeroCard
    {
        Title = title,
        Subtitle = description,
    };
    card.Buttons = new List<CardAction>
    {
        new CardAction(ActionTypes.OpenUrl, "Learn More",
value: url)
    };
    card.Images = new List<CardImage>
    {
        new CardImage( url = imageURL)
    };
    reply.Attachments.Add(card.ToAttachment());
    await context.PostAsync(reply);
}
```



```

    }
    else
    {
        await context.PostAsync("No good match found in KB.");
        int length = (((IMessageActivity)context.Activity).Text ??
string.Empty).Length;
        await context.PostAsync($"You sent
{((IMessageActivity)context.Activity).Text} which was {length}
characters");
    }
    context.Wait(MessageReceivedAsync);
}
}
}

```

Now that we are using the **RootDialog** to handle some logic for QnA Maker we need to modify our **QnaDialog** code a bit.

Between line 23-35 in the MessageReceivedAsync method, the last line on 33

Await context.PostAsync(answer);

Remove await and modify the code as:

context.Done(answer);

This way we can bring the answer back to the AfterQnA method to handle the reply.

```

public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> result)
{
    string safeText =
HttpUtility.UrlEncode(((IMessageActivity)context.Activity).Text);
    string answer = _QnAService.GetAnswer(safeText);
    if (string.IsNullOrEmpty(answer))
    {
        await context.PostAsync("No good match found in KB.");
    }
    else
    {
        await context.PostAsync(answer);
    }
}

```

Finally, now that we know our QnA Maker service works, modify the **MessagesController** to pass the activity back to the **RootDialog** to handle the conditions from here on out.

```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new
Dialogs.RootDialog());
    }
    else
    {
        HandleSystemMessage(activity);
    }
    var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
}
```

Restart the bot and emulator, and ask the latest question added to the knowledge base.

NOTE: Your emulator may report that your Bot code is not working. It may be an issue with your KB in which vertical bars have not been added to every answer or you may need to disable security and test on localhost. Visit **Web.config** and comment out your **MicrosoftAppId** and **MicrosoftAppPassword** for now to see if that fixes it.

```
<appSettings>
  <!--<add key="BotId" value="YourBotId" />-->
  <!--<add key="MicrosoftAppId" value="" />-->
  <!--<add key="MicrosoftAppPassword" value="" />-->
  <add key="QnAHost" value="ENTER_HOST_URL_HERE" />
  <add key="QnAEndPointKey" value="ENTER_ENDPOINT_KEY_HERE" />
  <add key="QnAKnowledgebaseId" value="ENTER_KNOWLEDGE_BASE_ID_HERE" />
</appSettings>
```

Documentation:

- <https://docs.microsoft.com/en-us/bot-framework/bot-service-troubleshoot-authentication-problems>

Awesome! We're now successfully using our QnA service to not only serve answers, but to store data which we can use in a meaningful way. As mentioned before, it is recommended that a consistent data format is used in the QnA service. The card was format defined in our bot was specific to the format we used in the knowledge base. If multiple data formats were defined, more logic would need to be implemented to handle differences in data formats the QnA service could respond with.

The screenshot displays the Bot Explorer application interface. On the left, a sidebar contains a 'BOT EXPLORER' section with a 'Rich Cards Bot' endpoint and a 'TRANSCRIPT EXPLORER' section. The main area shows a chat window titled 'Live Chat (Rich Cards Bot)'. The chat history includes a bot message: 'hello there, I am BIAD bot.' followed by a 'Learn More' button. A user message 'what is bot framework' is also visible. Below this, a card titled 'Bot Framework' is displayed, featuring a blue icon with a code symbol and a description: 'Bot Framework is a set of tools, services, connectors and Bot Builder SDKs. Developers can get started in seconds with out-of-the-box templates for scenarios including basic, form, language understanding, Q&A, and proactive bots.' This card also includes a 'Learn More' button. The chat window shows the bot's response at 10:56:32 PM. At the bottom, there is a text input field with a placeholder 'Type your message...' and a send button.

Extra Exercise

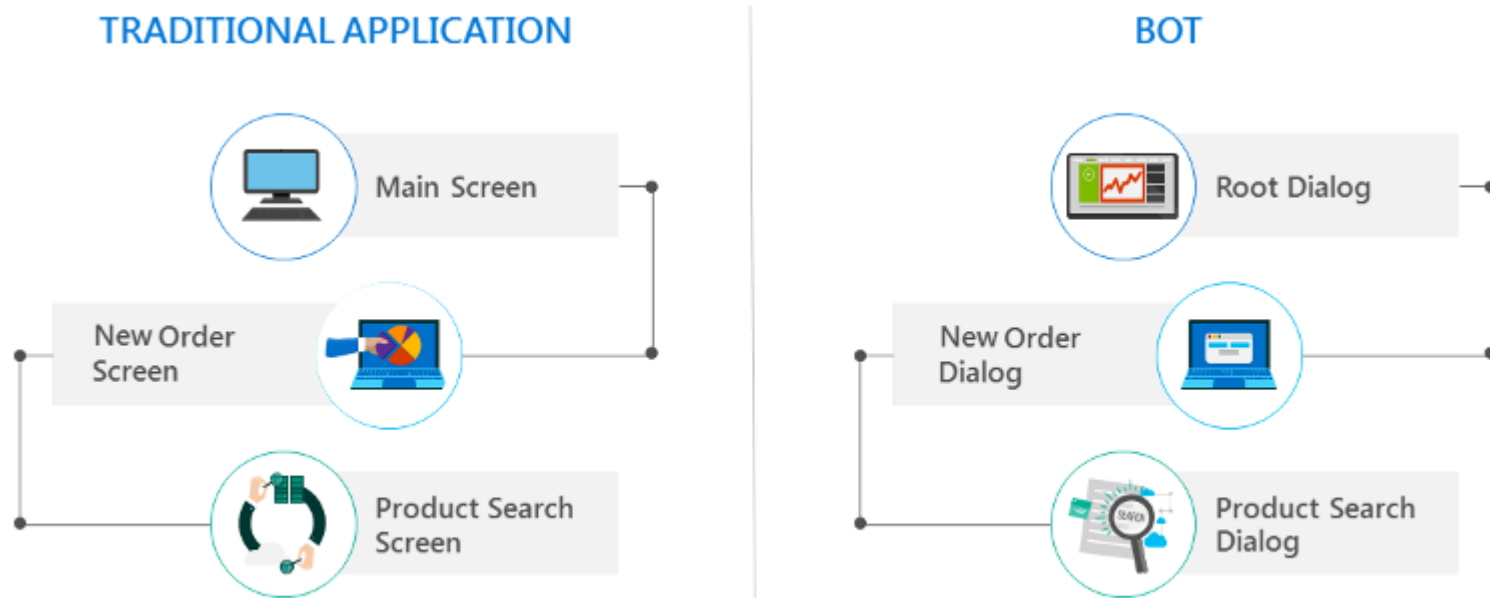
Your bot now only responds to in a specific format:

- E.g.: |description|
- E.g.: title|description|url|imageurl

Unfortunately, your bot now renders a card even if no other content exists. Your mission—should you choose to accept this challenge—time permitting, develop the logic to handle these exceptions and render an answer without a card when no specific format exists.

Lab 2 – Interacting with the user – Multiple Dialogs

This diagram shows the screen flow of a traditional application compared to the dialog flow of a bot.



In a traditional application, everything begins with the main screen. The main screen invokes the new order screen. The new order screen remains in control until it either closes or invokes other screens. If the new order screen closes, the user is returned to the main screen.

In a bot, everything begins with the root dialog. The root dialog invokes the new order dialog. At that point, the new order dialog takes control of the conversation and remains in control until it either closes or invokes other dialogs. If the new order dialog closes, control of the conversation is returned back to the root dialog.

This lab describes how to model this conversation flow by using dialogs and the Bot Builder SDK for .NET.

Learning Objectives

Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft's Bot Framework.

- Build a parent-child dialog hierarchy.
- Invoke child dialogs when conditions are met and take users back to the root dialog to handle the next activity.

Invoking multiples dialogs

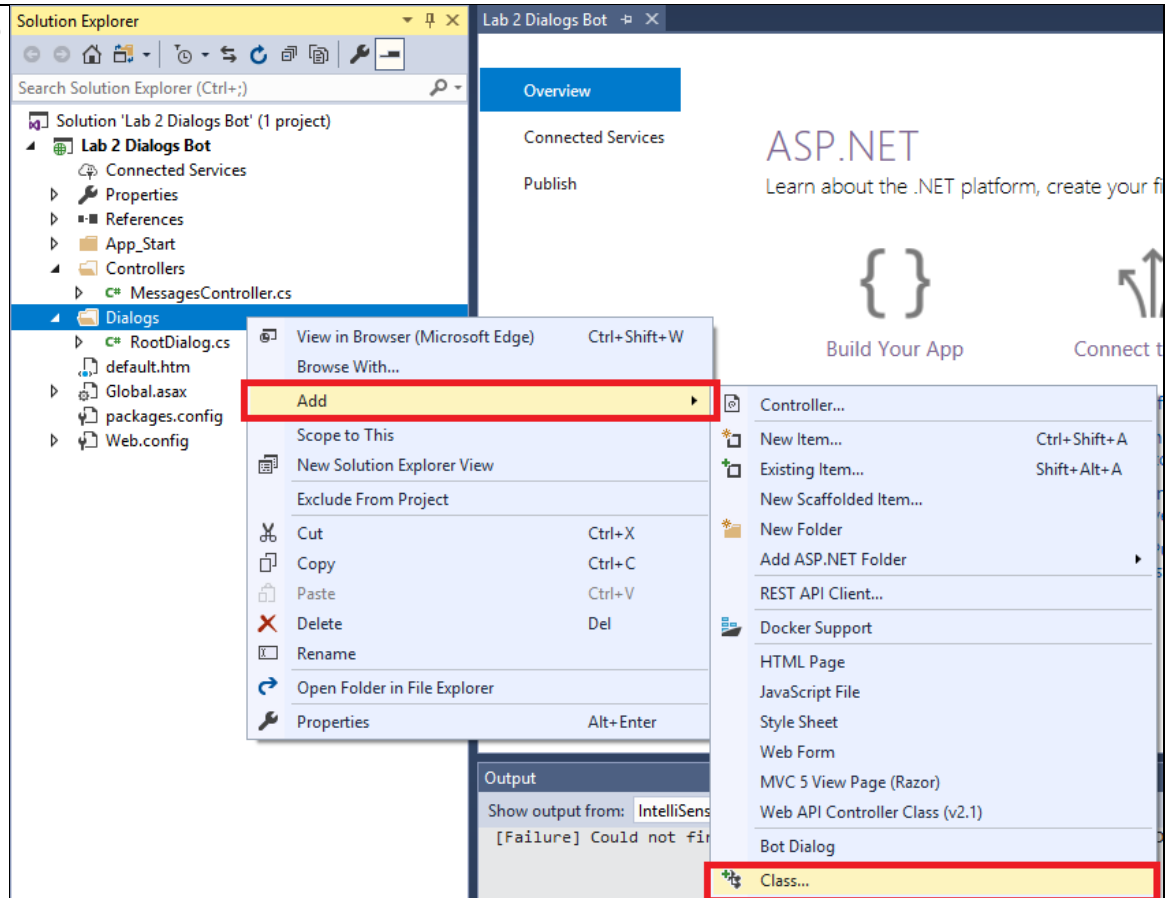
We can begin by setting up a new Bot template or working with the existing solution. If starting a new project, make sure to copy over the **RootDialog.cs** and **QnaDialog.cs** in your Dialogs folder as well as the **QnaMaker** C# class file and **Web.config** app settings configuration.

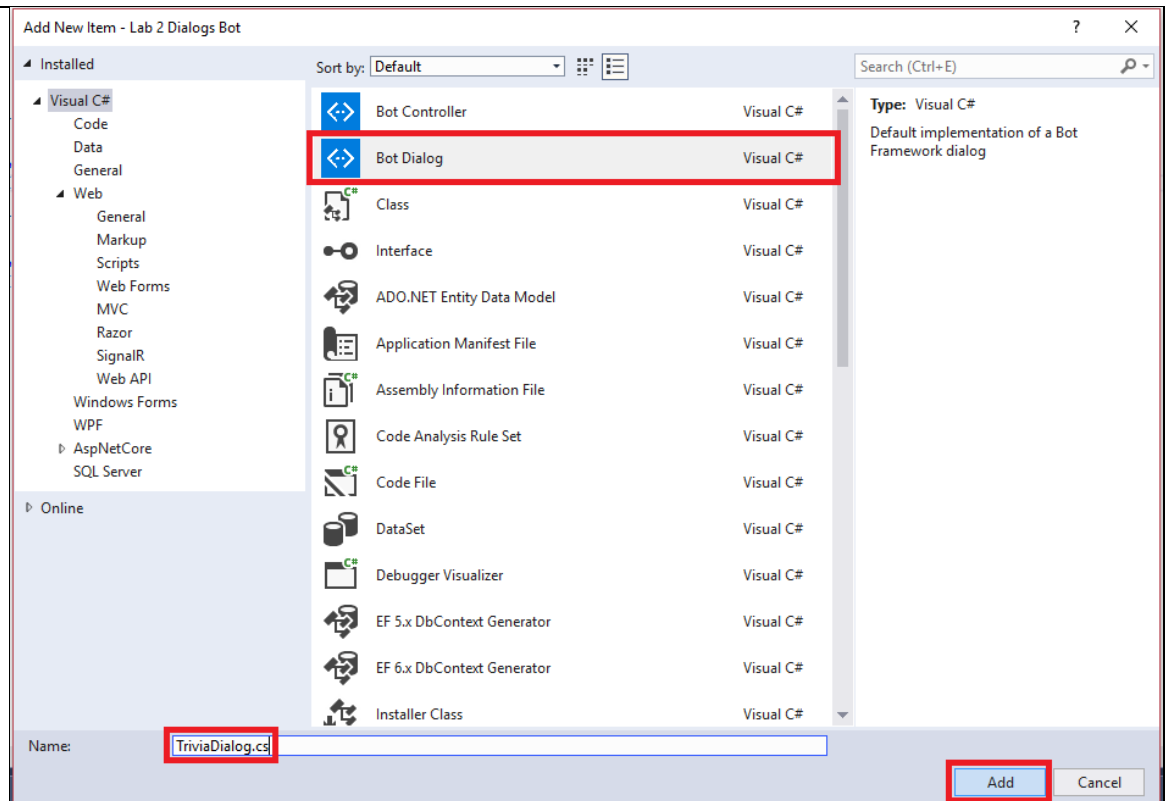
NOTE: Be careful to preserve the namespace!

You may need to install the RestSharp new NuGet package again.

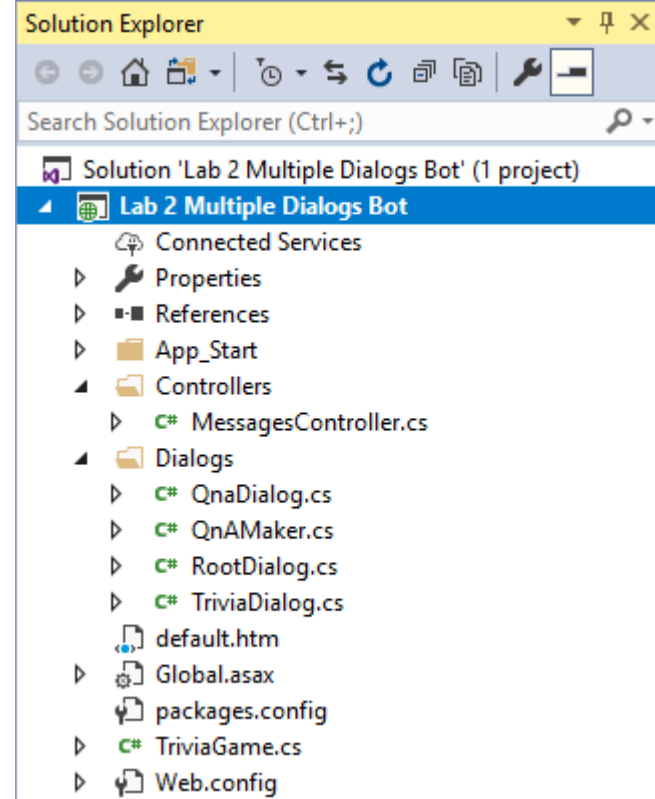
In this lab, we will modify the bot to have parent (**RootDialog**) and child dialogs. And pass along the message if the user response does not meet certain conditions.

1. Right-click on the Dialogs folder in Visual Studio and select Add, then Class....
2. Select Bot Dialog template and rename it to TriviaDialog.cs
3. Select Add.
4. Next, add another C# class titled TriviaGame.cs in the root folder of the solution.





NOTE: You should now have three/four dialogs: **RootDialog**, **QnaDialog**, & **TriviaDialog** and one new C# Class item called **TriviaGame** (and **QnAMaker** C# class depending on where you put it within your solution).



The **TriviaDialog.cs** looks like this→

NOTE: Code snippets! 😊

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace ENTER_SOLUTION_NAME_HERE.Dialogs
{
    [Serializable]
    public class TriviaDialog : IDialog<string>
    {
        private TriviaGame _game = null;

        public async Task StartAsync(IDialogContext context)
```

Start a game...

Welcome the user.

Post the question as a Choice Card.

Wait for Input from the user.

Reply back with the answer they chose.

If the answer is correct, say so.

Otherwise, say they're wrong.

Update the user with the current score.

```
{
    await context.PostAsync($"Welcome to Trivia, Let's play...");
    // post the question and choices as a hero card
    _game = new TriviaGame("");
    await context.PostAsync(_game.CurrentQuestion().Question);
    await context.PostAsync(MakeChoiceCard(context,
    _game.CurrentQuestion()));
    // wait for input
    context.Wait(MessageReceivedAsync);
}

/// <summary>
///     Here we'll check the user's answer and post the next
question until there are no more questions
/// </summary>
/// <param name="context">The current chat context</param>
/// <param name="result">The IAwaitable result</param>
/// <returns></returns>
private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
{
    var activity = (IMessageActivity)await result;
    await context.PostAsync($"You chose: {activity.Text}");
    int usersAnswer = -1;
    if (int.TryParse(activity.Text, out usersAnswer))
    {
        if (_game.Answer(usersAnswer))
        {
            await context.PostAsync("Correct!");
        }
        else
        {
            await context.PostAsync("Sorry, that's wrong :-(");
        }
        await context.PostAsync($"Your score is:
{_game.Score()}/{_game._questions.Count}. Next question!");
    }
}
```

Move to the next question.

Until all trivia questions have been answered...

Then, thank them for playing.

Else,

...if they enter any unexpected answer, tell the user as such.

This activity assembles the Choice Card for use in the trivia game.

Make sure to initialize the attachments to add buttons to the activity message.

Create a new Hero Card

For each CardAction, add a choice from the question to the actions (buttons). Later—index those choices.

Format the card, then post it back to the user.

```
TriviaQuestion nextQuestion = _game.MoveToNextQuestion();
if (nextQuestion != null)
{
    await context.PostAsync(nextQuestion.Question);
    await context.PostAsync(MakeChoiceCard(context,
nextQuestion));
    context.Wait(MessageReceivedAsync);
}
else
{
    await context.PostAsync("That's it! :-)");
    context.Done("");
}
}
else
{
    await context.PostAsync("I didn't quite get that, I am
only programmed to accept numbers :-(");
    context.Wait(MessageReceivedAsync);
}
}

private IMessageActivity MakeChoiceCard(IDialogContext context,
TriviaQuestion question)
{
    var activity = context.MakeMessage();
    // make sure the attachments have been initialized, we use the
attachments to add buttons to the activity message
    if (activity.Attachments == null)
    {
        activity.Attachments = new List<Attachment>();
    }

    var actions = new List<CardAction>();
    int choiceIndex = 0;
    foreach (string item in question.Choices)
```

The PostBack means the value will be sent back to the dialog as if the user typed it but it will be hidden from the chat window.

Add the Hero Card to "hold" the buttons and add it to the message activities attachments.

```
{
    actions.Add(new CardAction
    {
        Title = $"({choiceIndex}) {item}",
        Value = $"{choiceIndex}",
        Type = ActionTypes.PostBack // PostBack means the
        Value will be sent back to the dialog as if the user typed it but it will
        be hidden from the chat window
    });
    choiceIndex++;
}
// create a hero card to "hold" the buttons and add it to the
message activities attachments
activity.Attachments.Add(
    new HeroCard
    {
        Title = $"Choose One",
        Buttons = actions
    }.ToAttachment()
);

return activity;
}
}
```

The **TriviaGame.cs** looks like this →

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace ENTER_SOLUTION_NAME_HERE
{
    [Serializable]
    public class TriviaGame
    {
        private string _playersName;
        private int _currentQuestion = 0;
        private int[] _usersAnswers = new int[] { -1, -1, -1 };

        public List<TriviaQuestion> _questions = new List<TriviaQuestion>
        {
            new TriviaQuestion()
            {
                Index = 0,
                Answer = 3,
                Question = "How many pieces of contemporary art is in
Microsoft's collection?",
                Choices = new string[] { "0", "1000", "3000", "5000",
"10000"}
            },
            new TriviaQuestion()
            {
                Index = 1,
                Answer = 2,
                Question = "In 2016, Microsoft made a major breakthrough,
equaling that of humans, in what?",
                Choices = new string[] { "Writing Song Lyrics", "Derby Car
Racing", "Speech Recognition", "Predicting American Idol Winners"}
            },
            new TriviaQuestion()
        }
    }
}
```

```

        {
            Index = 2,
            Answer = 1,
            Question = "Annually, approximately how much money does
Microsoft spend on R&D?",
            Choices = new string[] { "$111 billion", "$11 billion", "$1
billion", "$1 dollar" }
        }
    };

    public TriviaGame(string playersName)
    {
        _playersName = playersName;
    }

    public TriviaQuestion CurrentQuestion()
    {
        return _questions.Where(q => q.Index ==
_currentQuestion).FirstOrDefault();
    }

    public TriviaQuestion MoveToNextQuestion()
    {
        _currentQuestion++;
        if (_currentQuestion < _questions.Count())
        {
            return CurrentQuestion();
        }
        else
        {
            _currentQuestion--;
            return null;
        }
    }

    public TriviaQuestion MoveToPreviousQuestion()
    {

```

```

        _currentQuestion--;
        if (_currentQuestion > 0)
        {
            return CurrentQuestion();
        }
        else
        {
            _currentQuestion = 0;
            return null;
        }
    }

    public TriviaQuestion MoveToFirstQuestion()
    {
        _currentQuestion = 0;
        return CurrentQuestion();
    }

    public bool Answer(int answer)
    {
        _usersAnswers[_currentQuestion] = answer;
        return _usersAnswers[_currentQuestion] ==
            _questions[_currentQuestion].Answer;
    }

    public int Score()
    {
        return _questions.Where(q => _usersAnswers[q.Index] ==
            q.Answer).Count();
    }
}

[Serializable]
public class TriviaQuestion
{
    public int Index { get; set; }
    public int Answer { get; set; }
}

```



```
public string Question { get; set; }  
public string[] Choices { get; set; }  
}  
}
```

Back in the **RootDialog**, we will create an **AfterTrivia** method; then, add extra condition in the **AfterQnA** method to handle the case if a user utters “trivia”.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;

namespace ENTER_SOLUTION_NAME_HERE.Dialogs
{
    [Serializable]
    public class RootDialog : IDialog<object>
    {
        public Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
            return Task.CompletedTask;
        }

        private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
        {
            var activity = await result as Activity;

            try
            {
                await context.Forward(new QnaDialog(), AfterQnA, activity,
CancellationTokens.None);
            }
            catch (Exception e)
            {
                // If an error occurred with QnA Maker Service, post it
out to the user
                await context.PostAsync(e.Message);

                // Wait for the next message from the user
                context.Wait(MessageReceivedAsync);
            }
        }
    }
}
```

This argument defines the IF statement when the message received contains the word "trivia," then call the **TriviaDialog**.

```
    }  
    }  
  
    private async Task AfterTrivia(IDialogContext context,  
IAwaitable<string> result)  
    {  
        await context.PostAsync("Thanks for playing!");  
        context.Wait(MessageReceivedAsync);  
    }  
  
    private async Task AfterQnA(IDialogContext context,  
IAwaitable<object> result)  
    {  
        string message = null;  
  
        try  
        {  
            message = (string)await result;  
        }  
        catch (Exception e)  
        {  
            await context.PostAsync($"QnAMaker: {e.Message}");  
            // Wait for the next message  
            context.Wait(MessageReceivedAsync);  
        }  
  
        // If the message summary - NOT_FOUND, then it's time to echo  
        if (message == null)  
        {  
            if (message.ToLowerInvariant().Contains("trivia"))  
            {  
                // Since we are not needing to pass any message to  
                // start trivia, we can use call instead of forward  
                context.Call(new TriviaDialog(), AfterTrivia);  
            }  
            else
```

In case you got stuck on the extra exercise after Lab 1_2 Rich Cards Bot, we provided some simple logic to handle the condition when the first element of the string is *null* or *empty*, trim the vertical bars and post the reply—otherwise render a card.

```
{
    // Otherwise, echo...
    await context.PostAsync($"You said: \"{message}\"");
    // Wait for the next message
    context.Wait(MessageReceivedAsync);
}
else
{
    // Display the answer from QnA Maker Service
    var answer = message;

    if (!string.IsNullOrEmpty(answer))
    {
        Activity reply =
((Activity)context.Activity).CreateReply();

        string[] qnaAnswerData = answer.Split('|');
        string title = qnaAnswerData[0];
        string description = qnaAnswerData[1];
        string url = qnaAnswerData[2];
        string imageURL = qnaAnswerData[3];

        if (title == "")
        {
            char charsToTrim = '|';
            await context.PostAsync(answer.Trim(charsToTrim));
        }

        else
        {
            HeroCard card = new HeroCard
            {
                Title = title,
                Subtitle = description,
            };
        }
    }
}
```

Here we've further modified the last condition to let the user down easier than an abrupt "No knowledge found" with an echo.

```
card.Buttons = new List<CardAction>
{
    new CardAction(ActionTypes.OpenUrl, "Learn More",
value: url)
};
card.Images = new List<CardImage>
{
    new CardImage( url = imageURL)
};
reply.Attachments.Add(card.ToAttachment());
await context.PostAsync(reply);
}
else
{
    await context.PostAsync("Sorry, I am having trouble
finding the answer to that. Try again later.");
}
context.Wait(MessageReceivedAsync);
}
}
}
```

Great, now we're movin!

Now, that you have implemented the Trivia Game, it is time to test it. How many answers can you get right?

This Trivia Game is so fun we want to know what other users think. So, we should take the opportunity ask them to fill out a survey and ask them how satisfied they are.

We will use the FormFlow tool to build a simple survey in a later lab.

However, before we do, we need a better way of surfacing this new feature to our users. One of the challenges in Conversational app development is how to "advertise" what experiences our bot can deliver to the end user without getting in the way. One of the best practices is to surface what services our bot can offer by proactively prompting the user.

Let's learn how to build adaptive cards to create this experience.

Lab 2.1 - Interacting with the user – Adaptive Cards

It is quite common for bots to interpret natural language. But natural language processing (NLP) can get quite tricky and extracting semantic representations in text is still an open research. A solution around avoiding large use of text is to leverage a rich User Interface (UI).

For example, you could ask a user “what kind of pizza would you like?”. This would result in a near infinite number of possible pizza varieties if you include all the combinations via free text. A more finite way of asking a user “what kind of pizza would you like?” is to present a list of options from which the user can select. This would avoid working with free text making the task of processing much easier for the Bot.

The aim of this lab is to demonstrate integration of user controls such as prompts, buttons and menus with bots.

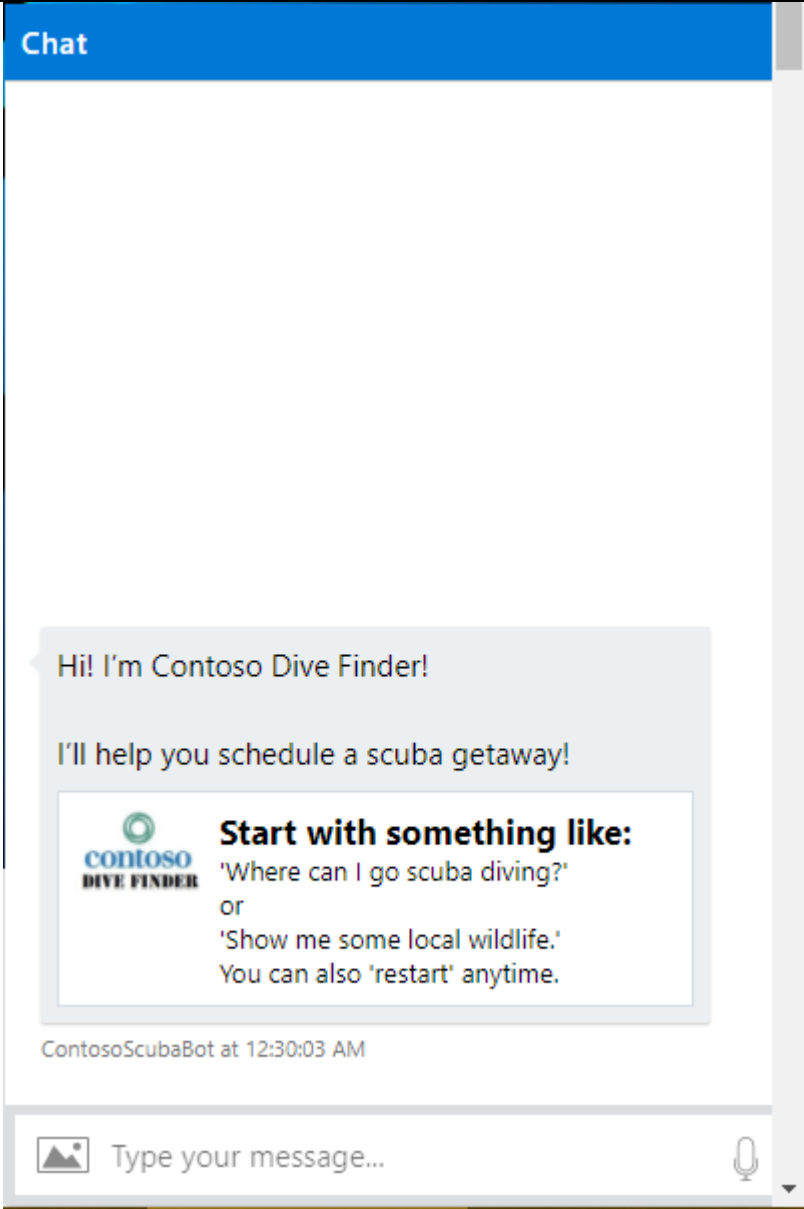
For this next lab,

Navigate to: <https://adaptivecards.io>

Read the documentation:

- <https://docs.microsoft.com/en-us/adaptive-cards/>

i Warning: Adaptive Cards is currently under development. While it is in preview, Microsoft is seeking feedback and says to expect a few rough edges.

Steps	Screenshots / Notes
<p>Adaptive cards are an open card exchange format enabling developers to exchange UI context in a common and consistent way.</p> <p>How they work: Card authors describe their content as a simple JSON object. That content can then be rendered natively inside a Host Application, automatically adapting to the look and feel of the Host.</p> <p>In this example, http://contososcubabot.azurewebsites.net/ the bot uses an active card to proactively provide the user with some suggestions on how to get started.</p> <p>With the Bot Framework you can write a single bot that is able to chat with users across multiple channels like Skype, Facebook Messenger, Slack, etc.</p> <p>You can navigate to https://adaptivecard.io and spend some time exploring the main rules and features of the JSON format in the schema explorer.</p> <p>The visualizer provides a helpful way to edit the JSON and visualize the object as you are editing.</p>	 <p>The screenshot shows a chat window titled "Chat". Inside, a message from "ContosoScubaBot" at 12:30:03 AM displays a proactive adaptive card. The card has a blue header with the "contoso DIVE FINDER" logo. The main content area is white and contains the text "Start with something like:" followed by two suggestions: "'Where can I go scuba diving?'" and "'Show me some local wildlife.'", separated by "or". Below these suggestions, it says "You can also 'restart' anytime." The card is set against a light blue background within the chat bubble.</p>

We need some way of proactively telling our user what we can do.

Let's build a card that tells them just that.

The **MyCard.json** example is available in your BIAD content folder, but customize it as you'd like:

The JSON to the right would build something like this →

BIAD Bot



Hello there!

Thanks for stopping by! I've got jokes and I've got smarts. Select and option below to get started, or type "start over" at any time to select a new option.

Tell me jokes

Tell me trivia

None of these

```
{
  "$schema": "http://adaptivecards.io/schemas/adaptive-card.json",
  "type": "AdaptiveCard",
  "version": "1.0",
  "body": [
    {
      "type": "Container",
      "items": [
        {
          "type": "TextBlock",
          "text": "BIAD Bot",
          "weight": "bolder",
          "size": "large"
        },
        {
          "type": "ColumnSet",
          "columns": [
            {
              "type": "Column",
              "width": "auto",
              "items": [
                {
                  "type": "Image",
```

```

        "url": "https://www.microsftevents.com/accounts/register123/microsoft/msft-v1/c-and-
e/events/mtc-155940/eventfiles/image.png",
        "size": "small",
        "style": "person"
    }
]
},
{
    "type": "Column",
    "width": "stretch",
    "items": [
        {
            "type": "TextBlock",
            "text": "Hello there!",
            "weight": "bolder",
            "size": "medium",
            "wrap": true
        },
        {
            "type": "TextBlock",
            "spacing": "none",
            "text": "",
            "isSubtle": true,
            "wrap": true
        }
    ]
}
]
}
]
},
{
    "type": "Container",
    "items": [
        {
            "type": "TextBlock",

```

```

        "text": "Thanks for stopping by! I've got jokes and I've got smarts. Select and option below to get
started, or type \"start over\" at any time to select a new option.",
        "wrap": true
    },
    {
        "type": "FactSet",
        "facts": [
            {
                "title": "",
                "value": ""
            },
            {
                "title": "",
                "value": ""
            },
            {
                "title": "",
                "value": ""
            },
            {
                "title": "",
                "value": ""
            }
        ]
    }
],
"actions": [
    {
        "type": "Action.Submit",
        "title": "Tell me jokes",
        "card": {
            "type": "AdaptiveCard",
            "body": [

```

```

        "type": "Input.Text",
        "id": "tellJoke",
        "title": "Tell me jokes"
    }
],
"actions": [
    {
        "type": "Action.Submit",
        "title": "OK"
    }
]
}
},
{
    "type": "Action.Submit",
    "title": "Tell me trivia",
    "card": {
        "type": "AdaptiveCard",
        "body": [
            {
                "type": "Input.Text",
                "id": "tellTrivia",
                "title": "Tell me trivia"
            }
        ],
        "actions": [
            {
                "type": "Action.Submit",
                "title": "OK"
            }
        ]
    }
}
},
{
    "type": "Action.Submit",
    "title": "None of these",

```

```
    "card": {
      "type": "AdaptiveCard",
      "body": [
        {
          "type": "Input.Text",
          "id": "doNothing",
          "title": "None of these"
        }
      ],
      "actions": [
        {
          "type": "Action.Submit",
          "title": "OK"
        }
      ]
    }
  ]
}
```

You will also need to right-click on project and manage NuGet Packages.

Search for Newtonsoft.Json and AdaptiveCards (v1.0.0 by AdaptiveCards).

Update and install packages.

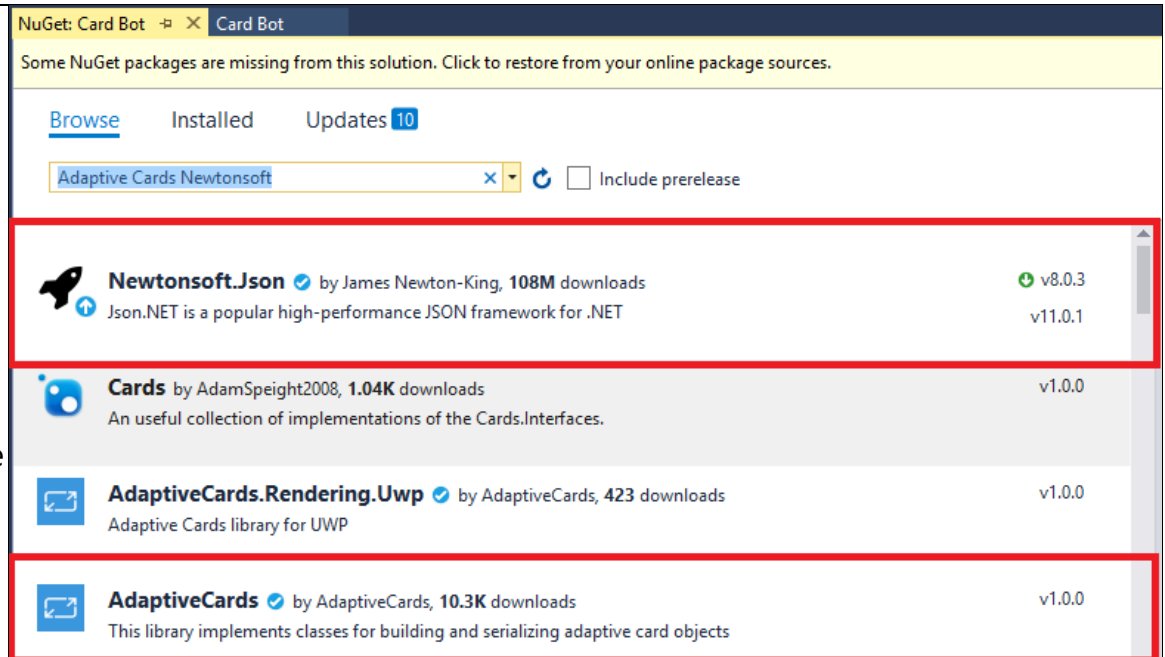
Next, you can add the JSON to a new folder called AdaptiveCards in the root folder of the project or directly to the root of the project like your **TriviaGame.cs**

But, since the path is local for the purpose of this lab, remember the path will be required to render the card properly.

NOTE: At the time of this writing, there has been significant breaking changes from previous versions.

- Package renamed from Microsoft.AdaptiveCards to AdaptiveCards
- All model classes have been prefixed with "Adaptive". E.g., TextBlock is now AdaptiveTextBlock

To see examples and read about more of these changes, visit: <https://docs.microsoft.com/en-us/adaptive-cards/create/libraries/net>



Now that **MyCard.json** is added to the project, in this exercise, we will learn how to parse an AdaptiveCard from JSON. This makes it easy to manipulate the object model or even render Adaptive Cards inside your app by using renderer SDKs.

Try adding this code snippet to replace your **HandleSystemMessage** activities in the **MessagesController.cs** to parse the JSON you created.

NOTE: You can use a local destination for the purpose of this exercise. In production the JSON files would most likely be hosted in Azure Blob Storage.

The bot is always added as a user of the conversation. Since, we don't want to display the adaptive card twice, ignore the second conversation update triggered by the bot.

Try to read the JSON from our newly created **MyCard.json**.

Use **Newtonsoft.JsonConvert** to deserialize the JSON into a C# Adaptive Card object.

Put the Adaptive Card as an attachment to the reply message.

```
private async Task HandleSystemMessage(Activity message)
{
    if (message.Type == ActivityTypes.DeleteUserData)
    {
        // Implement user deletion here
        // If we handle user deletion, return a real message
    }
    else if (message.Type == ActivityTypes.ConversationUpdate)
    {
        IConversationUpdateActivity update = message;
        using (var scope =
            DialogModule.BeginLifetimeScope(Conversation.Container, message))
        {
            var client = scope.Resolve<IConnectorClient>();
            if (update.MembersAdded.Any())
            {
                var reply = message.CreateReply();
                foreach (var newMember in update.MembersAdded)
                {
                    if (newMember.Name.ToLower() != "bot")
                    {
                        try
                        {
                            string json =
                                File.ReadAllText(HttpContext.Current.Request.MapPath("~/\\AdaptiveCards\\MyCard.json"));

                            AdaptiveCards.AdaptiveCard card =
                                JsonConvert.DeserializeObject<AdaptiveCards.AdaptiveCard>(json);
                            reply.Attachments.Add(new Attachment
                                {
                                    ContentType =
                                        AdaptiveCard.ContentType,

                                    Content = card
                                });
                        }
                        catch (Exception e)
                        {
                        }
                    }
                }
            }
        }
    }
}
```

If an error occurs, add the error text as the message.

```
        {
            reply.Text = e.Message;
        }
        await
client.Conversations.ReplyToActivityAsync(reply);
    }
}
}
}
}
else if (message.Type == ActivityTypes.ContactRelationUpdate)
{
}
else if (message.Type == ActivityTypes.Typing)
{
}
else if (message.Type == ActivityTypes.Ping)
{
}
}
```


Now that we have the JSON file in our project and it is properly populating at launch from the **MessagesController**, we need the **RootDialog** to understand a button click action.

Try to parse the JSON value to a string, where that token equals an "action"

In the case that the user clicks the "Tell me jokes" button, tell them I am currently working on this feature. Stay tuned.

In the case that the user clicks the "Play some trivia" button, call the **TriviaDialog** and start the game.

In any other case, report to the user that I do not know how to handle that action.

Add another case to handle the "None of these" button action.

```
private async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
{
    var activity = await result as Activity;

    try
    {
        if (activity.Value != null)
        {
            JToken valueToken =
                JObject.Parse(activity.Value.ToString());
            string actionValue = valueToken.SelectToken("action")
                != null ? valueToken.SelectToken("action").ToString() : string.Empty;

            if (!string.IsNullOrEmpty(actionValue))
            {
                switch
                (valueToken.SelectToken("action").ToString())
                {
                    case "jokes":
                        await context.PostAsync("Sorry, I'm
learning new jokes. Come back later.");
                        break;
                    case "trivia":
                        context.Call(new TriviaDialog(),
AfterTrivia);
                        break;
                    default:
                        await context.PostAsync($"I don't know how
to handle the action \"{actionValue}\".");
                        context.Wait(MessageReceivedAsync);
                        break;
                }
            }
        }
        else
        {

```

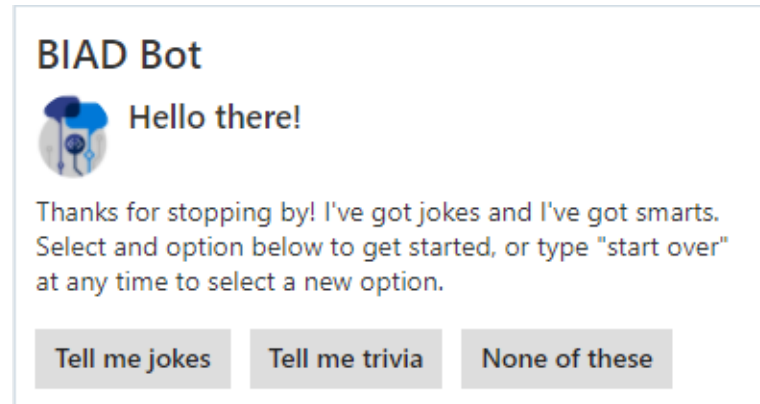
Otherwise, tell the user I am having trouble with my JSON (because I am in preview).

Else, try the **QnaDialog** for anything else and forward it to the **QnaDialog** if the threshold is met.

```
        await context.PostAsync("It looks like no \"data\"  
was defined for this. Check your adaptive cards JSON definition.");  
        context.Wait(MessageReceivedAsync);  
    }  
}  
else  
{  
    await context.Forward(new QnaDialog(), AfterQnA,  
activity, CancellationToken.None);  
}  
}  
catch (Exception e)  
{  
    // IF an error occurred with QnAMaker, post it out to the  
user  
    await context.PostAsync(e.Message);  
  
    // Wait for the next message from the user  
    context.Wait(MessageReceivedAsync);  
}  
}
```

Extra Exercise

Your bot now proactively prompts the user by showing an Adaptive Card to help the user get started. Your bot also manages the multiple dialogs by calling QnA Maker first; then, if no conditions are met, the message is sent back to the **RootDialog** to call additional dialogs. Your JSON card gives them a couple of options but we have not created a 'jokes' dialog.:



Create the JokesDialog and develop the additional logic in the Root Dialog to handle this.

One of the features of using Dialogs is that it encourages developers to define a conversational hierarchy.

This makes sense in some user scenarios where it is important to progress through a certain set of steps before reaching an outcome. It does, however have some limitations in that being explicit about the conversation hierarchy results in a conversation which is inflexible and often does not respond to the whim of the user. Scorable are a Bot Framework mechanism by which you can compose different parts of conversations without hardcoding the hierarchy. This composition allows a fluid user experience which is akin to a natural conversation.

The benefits of composing chatbot conversations in this way are:

- Users can access different parts of the conversation without knowing the route to get there.
- Users do not have to 'back track' conversations.

A developer wishing to implement scorable conversations should,

1. Provide 1 or more implementations of `Microsoft.Bot.Builder.Scorables.Internals.ScorableBase`. See <https://github.com/Microsoft/BotFramework-Samples/tree/master/blog-samples/CSharp/ScorableBotSample> for examples of this.
2. Make the Autofac IOC container aware of our scorable implementations. See `Global.asax.cs` for this.

In practice, the Bot Framework runtime will:

1. For each scorable implementation:
 - Call `HasScore()`
 - If `HasScore()` is true, then call `GetScore()`
2. Compare the results of `GetScore()` from each scorable implementation
3. Call `PostAsync()` on highest scorable
4. Call `DoneAsync()` on all scorables

Lab 3 – Building Forms with FormFlow and Language Understanding with Microsoft Cognitive Services

A FormFlow allows us to define a set of fields which the Bot Framework then handles building a dialog for us to collect data for each field. You can view the documentation (and a very good sandwich builder example) at the below link:

- <https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-formflow>

For our example, we will use a FormFlow to collect information regarding the user which will be requesting support from our bot. We would like to collect the following pieces of information:

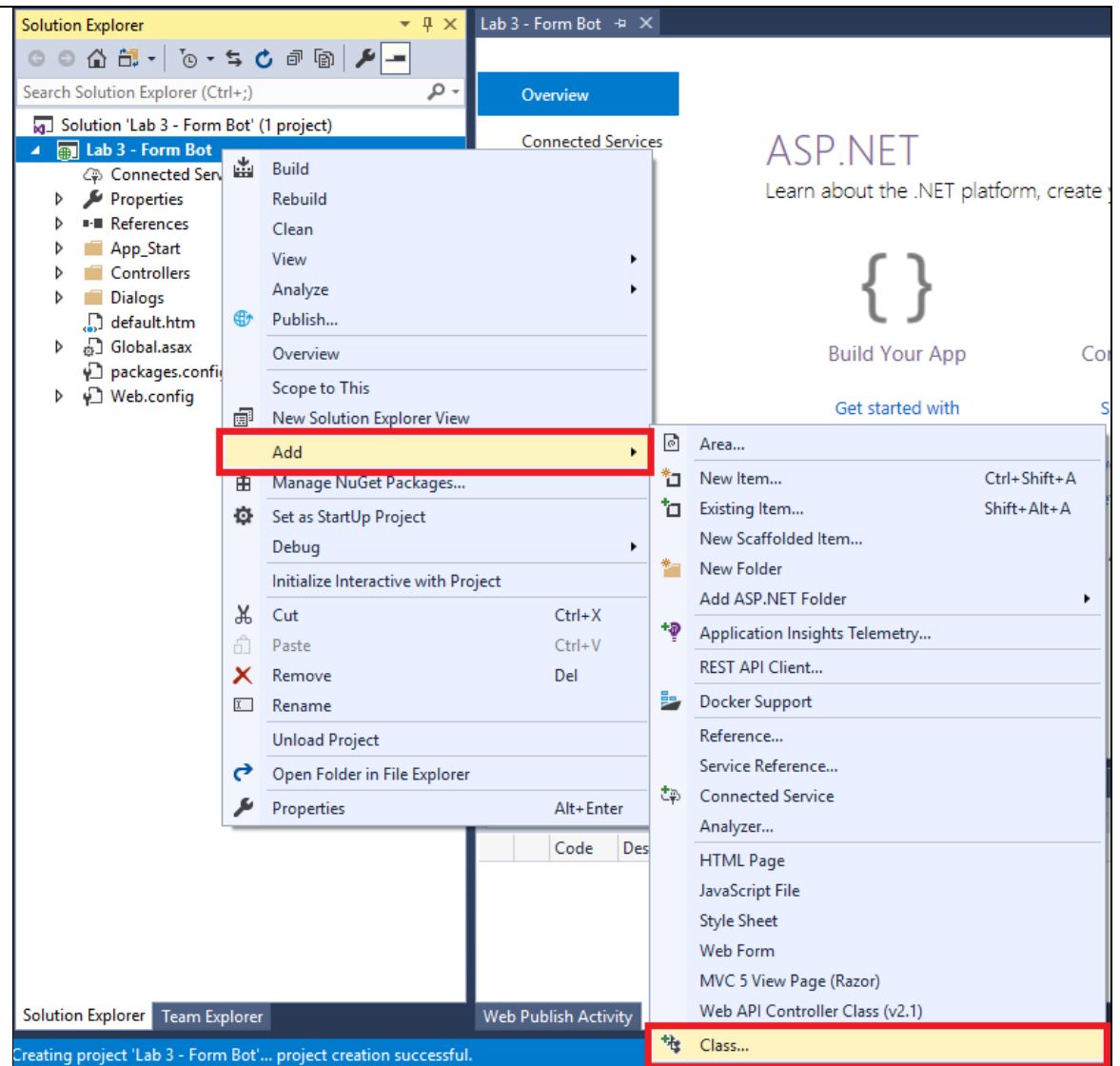
- Name
- Phone Number
- Email Address
- Department

To use a FormFlow, we first need to create a class with the fields defined we would like to collect.

Building a Form with FormFlow

Right-click on the project name in Visual Studio and select Add and then Class.

Then name it, **SurveyForm.cs**



Make sure to use the FormFlow directive then replace the boilerplate code with this →

The code is defining a **SurveyForm** class that contains data elements that the Bot Framework will collect. It asks for the fields in the order that they are defined in the class by default so it will start with **Name**, then **Phone Number**, and so on...

C# attributes can be used to provide guidance to the FormFlow regarding how to gather the data. We use two attributes here:

- Prompt – tells the FormFlow what text to use to request the data from the user. The {&} pattern in the Prompt string tells it to fill in the name of the field in that location. So, where it says, "Please enter your {&}.", for the Name field it will ask the user to "Please enter your name."
 - Pattern – tells the FormFlow to use a regular expression to validate the data entered. In this case, we are using regular expression to validate that the value entered for EmailAddress is truly a valid email address.
- Up to the challenge? – Add the logic to do the same for the PhoneNumber

Enums can be used in the **Department** field by declaring its data type to be **DepartmentOptions**.

```
using System;
using Microsoft.Bot.Builder.FormFlow;

namespace [ENTER_SOLUTION_NAME_HERE]
{
    public enum DepartmentOptions
    {
        Accounting,
        AdministrativeSupport,
        IT
    }

    [Serializable]
    public class SurveyForm
    {
        [Prompt("Please enter your {&}.")]
        public string Name;

        [Prompt("Please enter your {&}.")]
        public string PhoneNumber;

        [Prompt("Please enter your {&}.")]
        [Pattern(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}")]
        public string EmailAddress;

        [Prompt("What {&} do you work in? {||}.")]
        public DepartmentOptions? Department;

        public static IForm<SurveyForm> BuildForm()
        {
            return new FormBuilder<SurveyForm>().Build();
        }
    }
}
```

<p>The final piece for this class is to add a method (in this case called BuildForm) that uses the FormBuilder class to return a form built from the class as it was defined above.</p>	
<p>After your RootDialog attempts to take an action from the Adaptive Card we set up in the previous lab and scores the QnaDialog, the next method should be to ask users if they would like to take a survey after playing trivia.</p> <p>The method should look like this →</p> <p>The Boolean case moves along if the user says no.</p> <p>If the user says, yes, we call the SurveyDialog for them to fill out the form.</p> <p>We take them to AfterSurvey in the RootDialog once they are complete.</p> <p>NOTE: Use the lab solutions for reference.</p> <p>Finally, add this to the RootDialog →</p> <p>This should come directly after the previous method above.</p>	<pre>private async Task AfterTrivia(IDialogContext context, IAwaitable<string> result) { await context.PostAsync("Thanks for playing!"); PromptDialog.Confirm(context, AfterAskingAboutSurvey, "Would you like to take a survey?"); } private async Task AfterAskingAboutSurvey(IDialogContext context, IAwaitable<bool> result) { bool takeSurvey = await result; if (!takeSurvey) { context.Wait(MessageReceivedAsync); } else { var survey = new FormDialog<SurveyForm>(new SurveyForm(), SurveyForm.BuildForm, FormOptions.PromptInStart, null); context.Call<SurveyForm>(survey, AfterSurvey); } } private async Task AfterSurvey(IDialogContext context, IAwaitable<SurveyForm> result) { SurveyForm survey = await result; await context.PostAsync("Thanks for taking the survey!"); context.Wait(MessageReceivedAsync); }</pre>

That was simple!

Testing your bot, you should get a similar experience as seen here →

The survey will only be triggered **AfterTrivia**.

We have only scratched the surface, but as we've seen here, Dialogs are very powerful and flexible, but handling guided conversation such as ordering a sandwich can require a lot of effort. At each point in the conversation, there are many possibilities of what will happen next.

For example, you may need to clarify an ambiguity, provide help, go back, or show progress. By using FormFlow within the Bot Builder SDK for .NET, you can greatly simplify the process of managing a guided conversation like this.

Please enter your name.

Bot

Nick

User

Please enter your phone number.

Bot

555-555-5555

User

Please enter your email address.

Bot

nick@night.co

User

What department do you work in?.

Accounting

Administrative Support

IT

Bot

IT

User

Is this your selection?

- Name: Nick
- Phone Number: 555-555-5555
- Email Address: nick@night.co
- Department: IT

Bot at 11:56:33 PM

Yes

FormFlow automatically generates the dialogs that are necessary to manage a guided conversation, based upon guidelines that you specify. Although using FormFlow sacrifices some of the flexibility that you might otherwise get by creating and managing dialogs on your own, designing a guided conversation using FormFlow can significantly reduce the time it takes to develop your bot. Additionally, you may construct your bot using a combination of FormFlow-generated dialogs and other types of dialogs. For example, a FormFlow dialog may guide the user through the process of completing a form, while a LuisDialog may evaluate user input to determine intent.

That's what we plan to do next...

Basic features:

- <https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-formflow>

Advanced features:

- <https://docs.microsoft.com/en-us/bot-framework/dotnet/bot-builder-dotnet-formflow-advanced>

Sample of a simple sandwich bot:

- <https://github.com/Microsoft/BotBuilder/tree/master/CSharp/Samples/SimpleSandwichBot>

Lab 3.1 – Interpreting Emotional Sentiment using Language Understanding (LUIS) with Cognitive Services

Language Understanding service (LUIS) allows your application to understand what a person wants in their own words. LUIS uses machine learning to allow developers to build applications that can receive user input in natural language and extract meaning from it.

Learning Objectives

Upon completing this lab, you will have hands-on experience with the following functions and concepts related to Microsoft's Bot Framework.

- Integrate LUIS with a bot using .NET
- Build, train, test, and publish a LUIS app programmatically

A LUIS app is a domain-specific language model designed by you and tailored to your needs. You can start with a prebuilt domain model, build your own, or blend pieces of a prebuilt domain with your own custom information.

A model starts with a list of general user intentions such as "Book Flight" or "Contact Help Desk." Once the intentions are identified, you supply example phrases called utterances for the intents. Then you label the utterances with any specific details you want LUIS to pull out of the utterance.

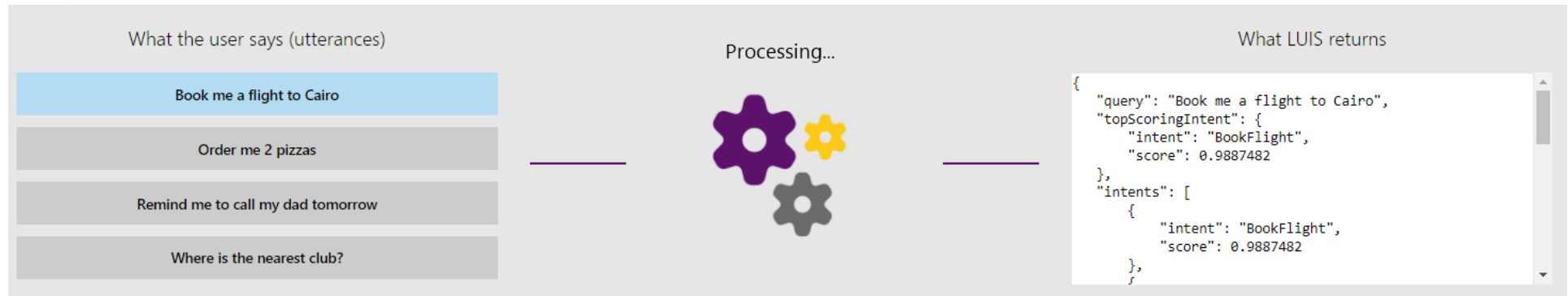
After the model is designed, trained, and published, it is ready to receive and process utterances. The LUIS app receives the utterance as an HTTP request and responds with extracted user intentions. Your client application sends the utterance and receives LUIS's evaluation as a JSON object. Your client app can then take appropriate action.

Key LUIS concepts

Intents - An intent represents actions the user wants to perform. The intent is a purpose or goal expressed in a user's input, such as booking a flight, paying a bill, or finding a news article. You define and name intents that correspond to these actions. A travel app may define an intent named "BookFlight."

Utterances - An utterance is text input from the user that your app needs to understand. It may be a sentence, like "Book a ticket to Paris", or a fragment of a sentence, like "Booking" or "Paris flight." Utterances aren't always well-formed, and there can be many utterance variations for a particular intent.

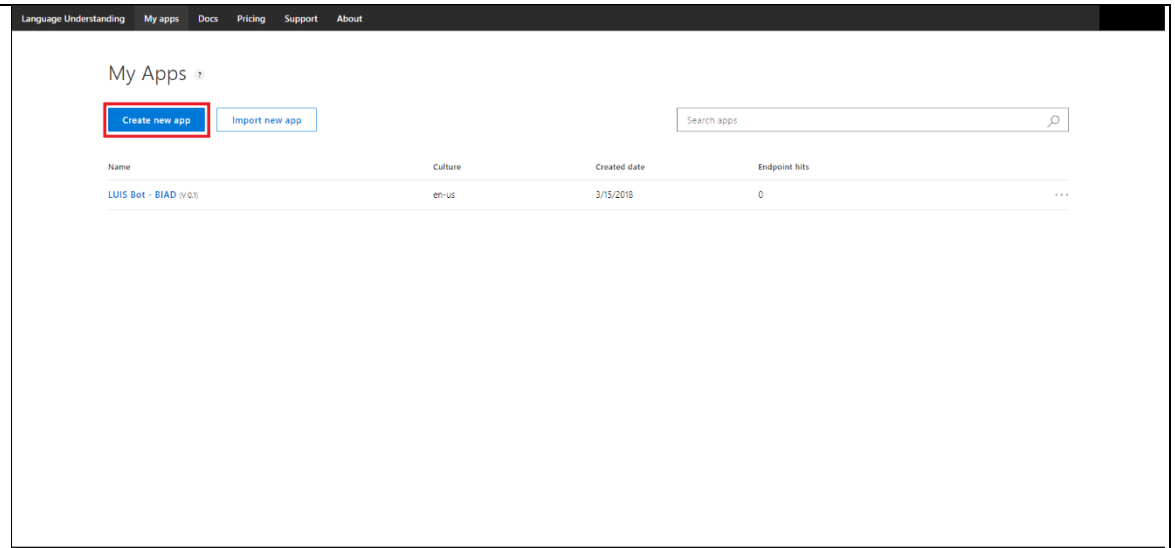
Entities - An entity represents detailed information that is relevant in the utterance. For example, in the utterance "Book a ticket to Paris", "Paris" is a location. By recognizing and labeling the entities that are mentioned in the user's utterance, LUIS helps you choose the specific action to take to answer a user's request.



<p>In this lab we will be adding one additional question to the survey. The aim of this question would be to measure the emotional sentiment of the user's statement after playing our trivia game. Essentially, we want to know how "positive", "negative", or "neutral" the user statement is and respond with a statement empathetic to that sentiment.</p> <p>e.g.: If the user said, "I got really bored with it.", we would want the bot to respond to the negative sentiment by saying something like, "I'm sorry to hear that. I will certainly try harder next time.</p> <p>Keep in mind this is our use, but the power of LUIS is that we can measure the emotional sentiment of any statement if we so desired.</p> <p>For a customer-facing bot, if a conversation is turning negative, we can have a human intercept the conversation before a customer is lost.</p>	
---	--

So, we will start this lab by navigating to:
<https://luais.ai>

Then, select 'Create new app'



Next, let's give it an appropriate name...

... and description.

Create new app

Name (Required)

Lab 3 LUIS Bot

Culture (Required)

English

** Culture is the language that your app understands and speaks, not the interface language.

Description

This bot will analyze the emotional sentiment of the user satisfaction.

Done

Cancel

You will notice the LUIS app comes with a pre-defined intent "None".

In the bottom-left you can explore the more than 20 pre-built domains LUIS already has domain expertise in to help you build more advanced language understanding in your applications.

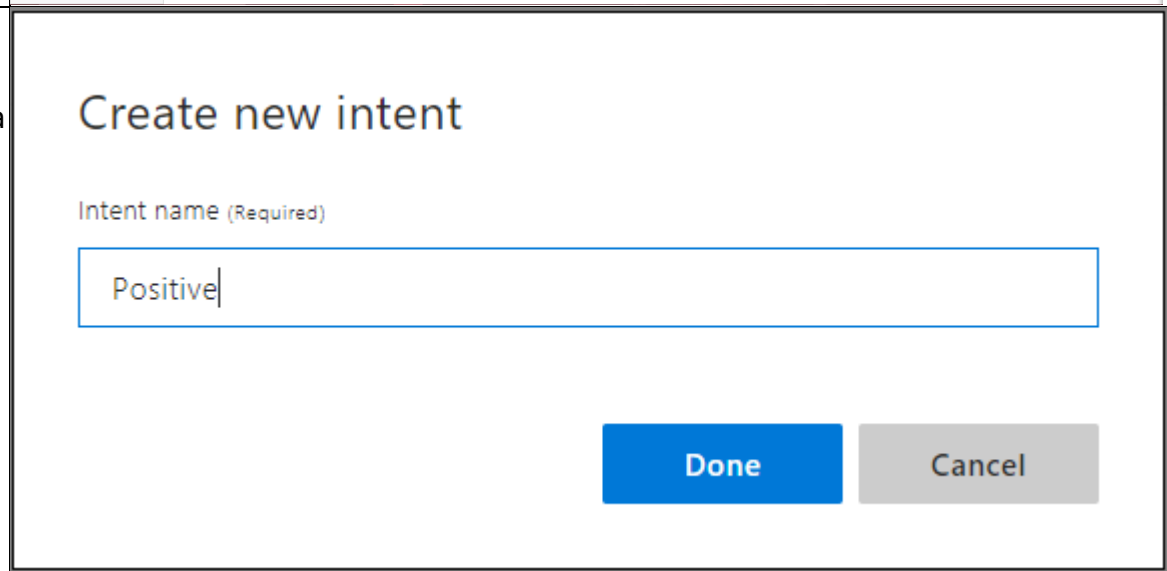
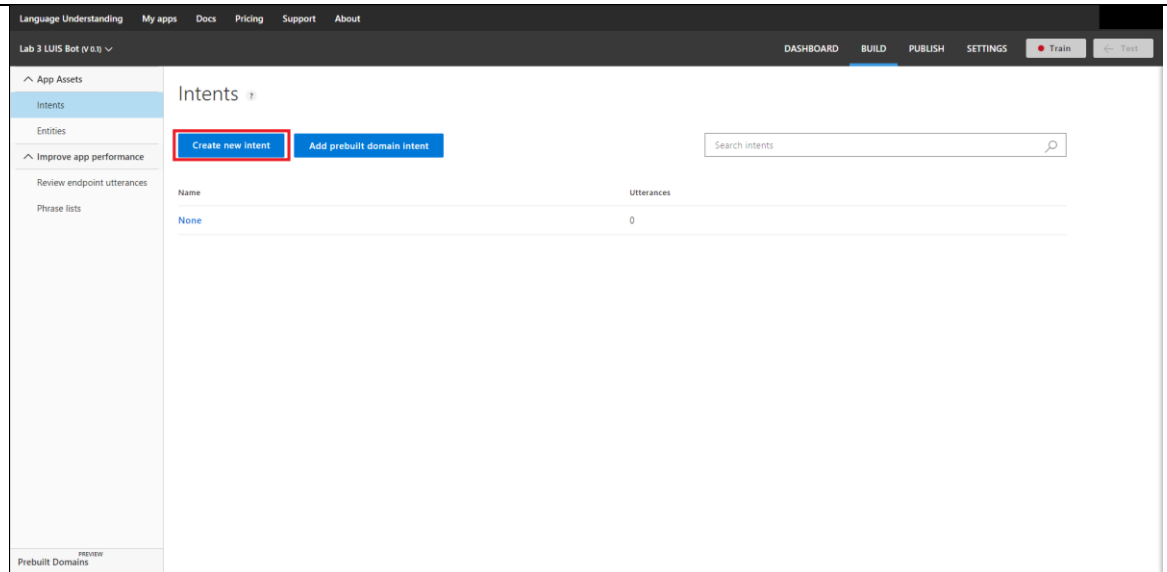
This is the same language understanding engine that powers Cortana!

For now, we will select "Create new intent."

Begin by adding the first intent.

For the purpose of this lab, we will be adding a "positive", "neutral", and "negative."

But, if you are up to the challenge you are certainly welcome to build a more sophisticated language model.



Some examples of utterances with positive intent are:

- I am happy
- I like it
- It was interesting
- It was entertaining
- I am satisfied
- I enjoyed it
- It was fun
- I learned something
- I liked it a lot

Just be sure to add at least 5 utterances, like QnA Maker, you can always go back and add more later.

To save time, you can import the **LUIS Bot – BIAD.json** and explore this later.

Positive

[Delete Intent](#)

Type about 5 examples of what a user might say to trigger this task and hit Enter.


[Reassign intent](#)[Delete utterance\(s\)](#)

Filters: ☐ Errors ☒ Entity ☒ Entities view ☐ Prefix search


☐ Utterance

Labeled intent 


☐ i am happy

Positive 0.82 


[...](#)☐ i like it

Positive 0.92 


[...](#)☐ it was interesting

Positive 0.89 


[...](#)☐ it was entertaining

Positive 0.89 


[...](#)☐ i am satisfied

Positive 0.87 


[...](#)☐ i enjoyed it

Positive 0.91 


[...](#)☐ it was fun

Positive 0.88 

[...](#)☐ i learned something

Positive 0.89 

[...](#)☐ i liked it a lot

Positive 0.9 

[...](#)

Now it is time to test train and publish your LUIS app.

You can test utterances in the Test window, and re-assign intent if you receive intents from utterances you so desire.

Keep in mind, every time you add a new utterance you must retrain and re-publish your app in order to experience the changes.

The screenshot displays the Microsoft LUIS web interface. At the top, there is a navigation bar with links for 'apps', 'Docs', 'Pricing', 'Support', and 'About'. Below this is a secondary navigation bar with tabs for 'DASHBOARD', 'BUILD', 'PUBLISH', and 'SETTINGS'. To the right of these tabs are two buttons: 'Train' (with a green dot icon) and 'Test' (with a blue arrow icon). The main content area is titled 'Positive' with a pencil icon for editing. It features a text input field with the placeholder 'Type about 5 examples of what a user might say'. Below this is a search bar labeled 'Search for an utterance' with a magnifying glass icon. To the right of the search bar is a 'Reassign' button. Underneath the search bar, there are filter options: 'Filters: ☐ Errors' and 'Entity ☐'. A blue toggle switch is also visible. Below the filters, there is a list of utterances, each with a checkbox and the text: 'Utterance', 'i am happy', and 'i like it'. On the right side of the interface, a 'Test' panel is open. It has a 'Start over' link and a 'Batch testing panel' link. A text input field in the 'Test' panel contains the text 'that was fun'. Below the input field, the result is displayed as 'Positive (0.64)' with a blue 'Inspect' link. A dark grey arrow points from the input field to the result.

Once your LUIS app is trained and up to date, it is now time to publish.

The screenshot displays the Microsoft LUIS portal interface. At the top, there is a navigation bar with links for 'apps', 'Docs', 'Pricing', 'Support', and 'About'. Below this, a secondary navigation bar contains 'DASHBOARD', 'BUILD', 'PUBLISH', and 'SETTINGS'. The 'BUILD' tab is currently selected. In the top right corner of the 'BUILD' section, a 'Training ...' button is highlighted with a red rectangular box. To its right is a blue 'Test' button. Below the navigation bar, a status bar indicates 'App up to date'. The main content area is titled 'Positive' with an edit icon. It features a text input field with the placeholder 'Type about 5 examples of what a user might say'. Below this is a search bar labeled 'Search for an utterance' with a magnifying glass icon. A 'Filters' section includes checkboxes for 'Errors' and 'Entity' (which is selected), and a toggle switch. Below the filters, there are three example utterances, each with a checkbox: 'Utterance', 'i am happy', and 'i like it'. On the right side of the interface, a 'Test' panel is visible, showing a 'Start over' link, a 'Batch testing panel' link, a text input field for 'Type a test utterance', and a sample utterance 'that was fun' with a score of 'Positive (0.64)' and an 'Inspect' link.

Finally, we can publish this LUIS app to our staging area.

Be sure to set Publish to: "Staging"

Select the right time zone so the Cognitive Services deploys in the datacenter nearest to you to avoid latency.

Select 'Publish to staging slot'

Your resources will be share with you in the format shown →

These resources are required, just like QnA Maker in order to work in your bot.

Publish app ?

Published version: 0.1

Published date: Mar 19, 2018, 12:18:50 AM (1 hour(s) ago)

Publish to

Timezone:

- ☒ Include all predicted intent scores ?
☐ Enable Bing spell checker ?

Publish to staging slot

Resources and Keys

Add Key

☒ North America Regions ☐ South America Regions ☐ Europe Regions ☐ Asia Regions ☐ Australia Regions

Resource Name	Region	Key String	Endpoint
Starter_Key	westus	Subscription Key	https://westus.api.cognitive.microsoft.com/luis/v2.0/apps/-i&staging=true&verbose=true&timezoneOffset=-480&q=
		Subscription Key	Application Id ?subscription-

Connect LUIS to your bot

Now it's time to go back into our bot code.

So, we will start by adding a LuisDialog class in the Dialogs folder.

Replace your boilerplate dialog code with this
→

We will explicitly state we will return a string using context.Done(string)

We will Setup the Luis Service in this dialog and the DispatchToIntentHandler will receive the message and return the best intent as a result.

Once the message is received,

...we will set up and values of the LuisModelAttribute that you cannot set in the constructor

The LUIS service to use in the base constructor of this **LuisDialog**

The code checks to see if Debug is set to true

(This allow us to test our code in debug while it is staged until we are ready to go the production.)

```
namespace [ENTER_SOLUTION_NAME_HERE].Dialogs
{
    [Serializable]
    public class LuisDialog : LuisDialog<string> // explicitly state we
    will return a string using context.Done(string)
    {

        public LuisDialog() : base(SetupLuisService())
        {

        }

        protected override Task DispatchToIntentHandler(IDialogContext
context, IAwaitable<IMessageActivity> item, IntentRecommendation
bestIntent, LuisResult result)
        {
            return base.DispatchToIntentHandler(context, item, bestIntent,
result);
        }

        protected override Task<string>
GetLuisQueryTextAsync(IDialogContext context, IMessageActivity message)
        {
            return base.GetLuisQueryTextAsync(context, message);
        }

        protected override Task MessageReceived(IDialogContext context,
IAwaitable<IMessageActivity> item)
        {
            return base.MessageReceived(context, item);
        }

        public static LuisService SetupLuisService()
        {
            LuisModelAttribute attribute = new LuisModelAttribute(
                ConfigurationManager.AppSettings["LuisAppId"],
```

Return the attribute from LUIS...

If "Positive", then say this...

Then go back to the **RootDialog**.

If "Negative", then say this...

Then go back to the **RootDialog**.

If "Neutral", then say this...

```
        ConfigurationManager.AppSettings["LuisAPIKey"],
        domain:
ConfigurationManager.AppSettings["LuisAPIHostName"]);
#if DEBUG
        attribute.Staging = true;
#endif
        return new LuisService(attribute);
    }

    [LuisIntent("Positive")]
    public async Task HappyIntent(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("I'm happy you're happy. I'm happy too.
We're both happy.");

        context.Done("");
    }

    // Should be invoked when the user asks for help with no specific
information
    // such as "what can you help me with?"
    [LuisIntent("Negative")]
    public async Task SadIntent(IDialogContext context, LuisResult
result)
    {
        await context.PostAsync("Oh man, that's too bad. You should be
happy like me, it's much better!");
        context.Done("");
    }

    // Should be invoked when the user asks something generic and
includes a recognized application name
    // such as "what can you do in SDMPPlus?"
    [LuisIntent("Neutral")]
```

If the Intent attribute is not defined or "None"...

Then, don't say anything and go back to the **RootDialog**.

```
public async Task NuetralIntent(IDialogContext context, LuisResult
result)
{
    await context.PostAsync("Yea, yea, it is what it is.");
}

[LuisIntent("")]
[LuisIntent("None")]
public async Task NoneIntent(IDialogContext context, LuisResult
result)
{
    context.Done("none");
}
}
```

Next, we will add another dialog class called **AskLuis.cs**

The code will look like this →

This code essentially forwards the message sent by the user straight to LUIS for scoring.

We don't have to do any casting because we explicitly said we will return a string in **LuisDialog**

If not intent was found, don't say anything.

```
namespace [ENTER_SOLUTION_NAME_HERE].Dialogs
{
    [Serializable]
    public class AskLuis : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            context.Wait(MessageReceivedAsync);
        }

        public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<object> result)
        {
            IMessageActivity activity = (IMessageActivity)await result;
            await context.Forward(new LuisDialog(), AfterLuis, activity,
CancellationToken.None);
        }

        public async Task AfterLuis(IDialogContext context,
IAwaitable<string> message)
        {
            string val = await message;
            if (val == "none")
            {
                context.Wait(MessageReceivedAsync);
            }
            else
            {
                context.Done("");
            }
        }
    }
}
```


Finally, in the RootDialog , after AfterAskingAboutSurvey and before AfterQnA , supply this code in between →	<pre>private async Task AfterSurvey(IDialogContext context, IAwaitable<SurveyDialog> result) { SurveyDialog survey = await result; await context.PostAsync("Thanks for filling out the form! One last question. Please, describe how the game made you feel."); context.Call(new AskLuis(), AfterAskLuis); } private Task AfterAskLuis(IDialogContext context, IAwaitable<object> result) { context.Wait(MessageReceivedAsync); return Task.CompletedTask; }</pre>
And, guess what... you're done!	

Testing your bot should demonstrate all of the objectives of today's workshop including, but not limited to:

- Developing bots in .NET using the Application template.
- Testing a bot using the Bot Emulator.
- Connecting QnAMaker from Microsoft Cognitive Services to your bot to answer common questions.
- Build rich cards in .NET to surface buttons and linkable content.
- Manage conversation flow with multiple dialogs
- Create rich, Adaptive Cards to create a similar, engaging experience to the end user across many different channels
- Create forms using FormFlow
- Use Language Understanding from LUIS by Microsoft Cognitive Services to analyze emotional sentiment and infuse AI in your bot.

We hope you enjoyed the workshop as much as we enjoyed making it. Congratulations, and thanks for attending Bot in a Day!

By using this demo/lab, you agree to the following terms:

The technology/functionality described in this demo/lab is provided by Microsoft Corporation for the purposes of obtaining your feedback and to provide you with a learning experience. You may only use the demo/lab to evaluate such technology features and functionality and provide feedback to Microsoft. You may not use it for any other purpose. You may not modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell this demo/lab or any portion thereof.

COPYING OR REPRODUCTION OF THE DEMO/LAB (OR ANY PORTION OF IT) TO ANY OTHER SERVER OR LOCATION FOR FURTHER REPRODUCTION OR REDISTRIBUTION IS EXPRESSLY PROHIBITED.

THIS DEMO/LAB PROVIDES CERTAIN SOFTWARE TECHNOLOGY/PRODUCT FEATURES AND FUNCTIONALITY, INCLUDING POTENTIAL NEW FEATURES AND CONCEPTS, IN A SIMULATED ENVIRONMENT WITHOUT COMPLEX SET-UP OR INSTALLATION FOR THE PURPOSE DESCRIBED ABOVE. THE TECHNOLOGY CONCEPTS REPRESENTED IN THIS DEMO/LAB MAY NOT REPRESENT FULL FEATURE FUNCTIONALITY AND MAY NOT WORK THE WAY A FINAL VERSION MAY WORK. WE ALSO MAY NOT RELEASE A FINAL VERSION OF SUCH FEATURES OR CONCEPTS. YOUR EXPERIENCE WITH USING SUCH FEATURES AND FUNCTIONALITY IN A PHYSICAL ENVIRONMENT MAY ALSO BE DIFFERENT.

FEEDBACK: If you give feedback about the technology features, functionality and/or concepts described in this demo/lab to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You may also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in the. These rights survive this agreement.

MICROSOFT CORPORATION HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DEMO/LAB, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. MICROSOFT DOES NOT MAKE ANY ASSURANCES OR REPRESENTATIONS WITH REGARD TO THE ACCURACY OF THE RESULTS, OUTPUT THAT DERIVES FROM USE OF DEMO/LAB, OR SUITABILITY OF THE INFORMATION CONTAINED IN THE DEMO/LAB FOR ANY PURPOSE.

DISCLAIMER

This demo/lab contains only a portion of new features and enhancements in the Microsoft Bot Framework, Microsoft Cognitive Services, and the Azure Bot Service. Some of the features might change in the future releases of these products. In this demo/lab, you will learn about some, but not all, new features.