+Creating a bot using the Microsoft Bot Framework
In C#

Hands-on Lab Manual

Table of Contents

+Creating a bot using the Microsoft Bot Framework In C# 1
Hands-on Lab Manual 1
Lab Introduction 3
Objectives 3
Prerequisites 3
Lab Scenarios 3
Configuration and Setup 4
Copy/Paste of Code 9
Exercise 1: Basic Bot using BotBuilder 10
Exercise 2: Creating Dialogs 16
Exercise 3: Form Flow 29
Exercise 4: Carousels, Cards, and Adaptive cards 45
Exercise 5: Using Intent Dialogs (LUIS) 60
Additional Resources 69
Copyright 70

Lab Introduction

Objectives

After completing these self-paced labs, you will be able to:

- Have an understanding of the basics of the Bot Framework

Prerequisites

- Visual Studio 2015 (community edition or other editions)
- NGrok
- Bot Application Template
- Basic understanding of C#

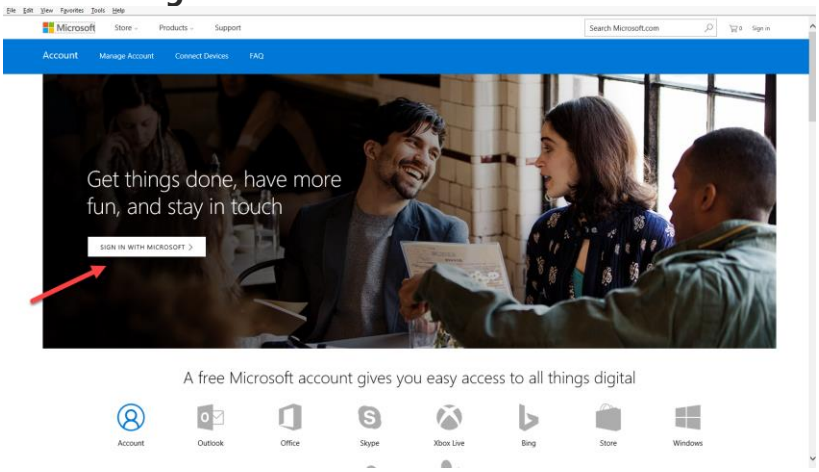
Lab Scenarios

This series of exercises is designed to show you how to get started using the Microsoft Bot Framework. In this lab, we are going to create a DinnerBot that will allow you to make reservations for a restaurant.

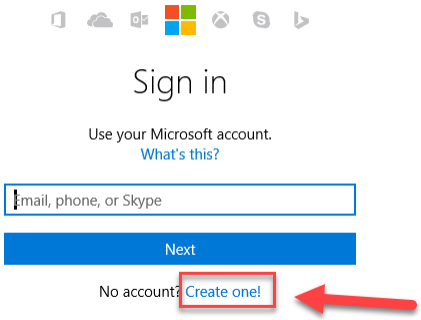
Configuration and Setup

1. Install prerequisite software
    - **Visual Studio 2015** : https://www.visualstudio.com/vs/community/
    - **NGrok** : https://ngrok.com/
      Skype : http://skype.com (if you want to test a Skype Bot)
    - **C# Bot Application Template**: http://aka.ms/bf-bc-vstemplate When this zip is downloaded, copy (not unzipped) to %USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#
    - **Update all Visual Studio Extensions (Tools → Extensions and Updates →Updates)**
    - **Bot Framework Emulator:** https://docs.microsoft.com/en-us/bot-framework/resources-tools-downloads
    - **Create a Microsoft ID** (if you don't already have one)

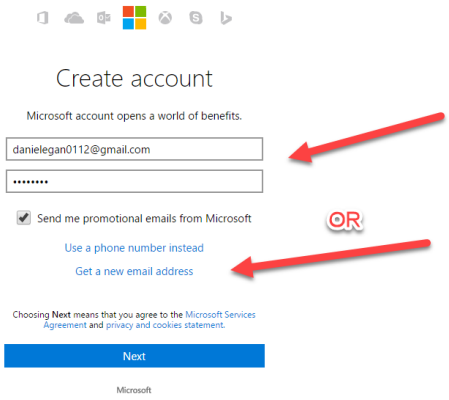
      Go to the Microsoft account sign-up page https://account.microsoft.com/ and **click Sign In with Microsoft**.



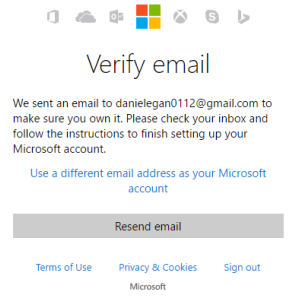
    - Click on the **Create One** link.

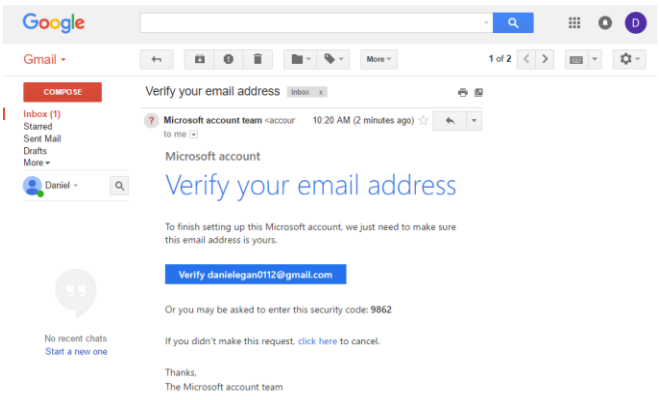


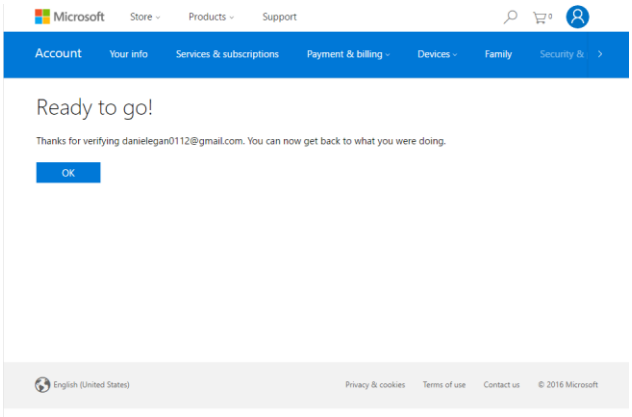
- In the User name box enter your existing email address, or click Get a new email address to create an Outlook or Hotmail address.



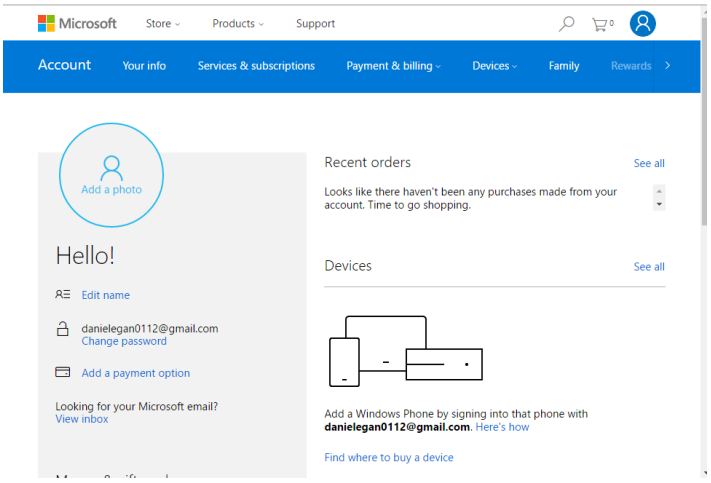
NOTE: **If you use an existing email address you will need to verify it before moving on.**



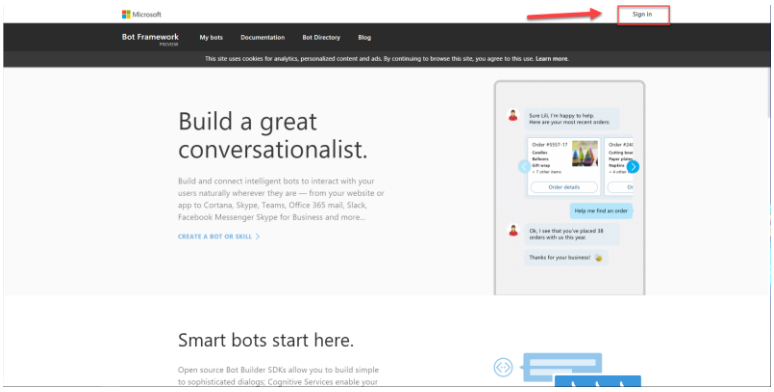




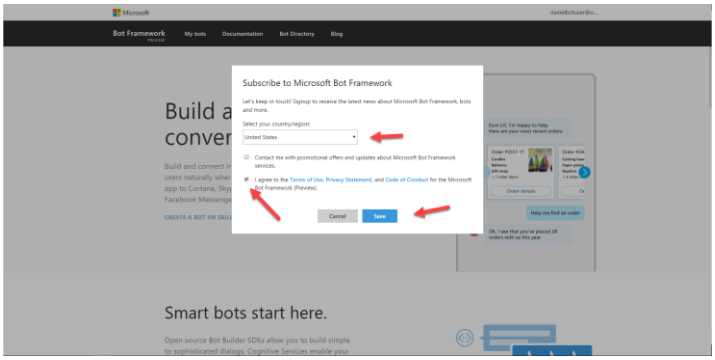
- Either path will take you to this screen

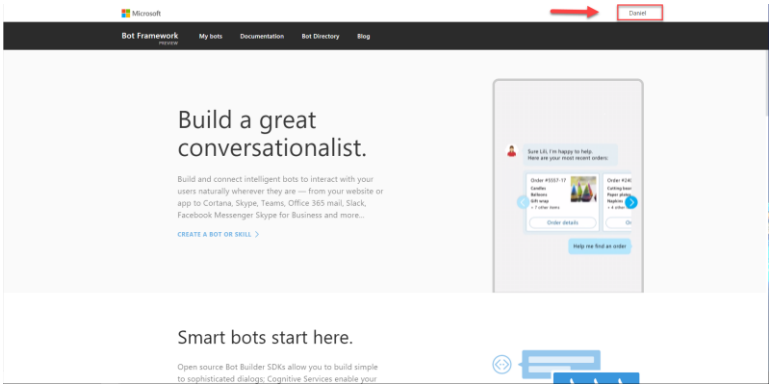


2. Create a BotFramework account
   - Navigate to http://BotFramework.com
   - Click on sign in



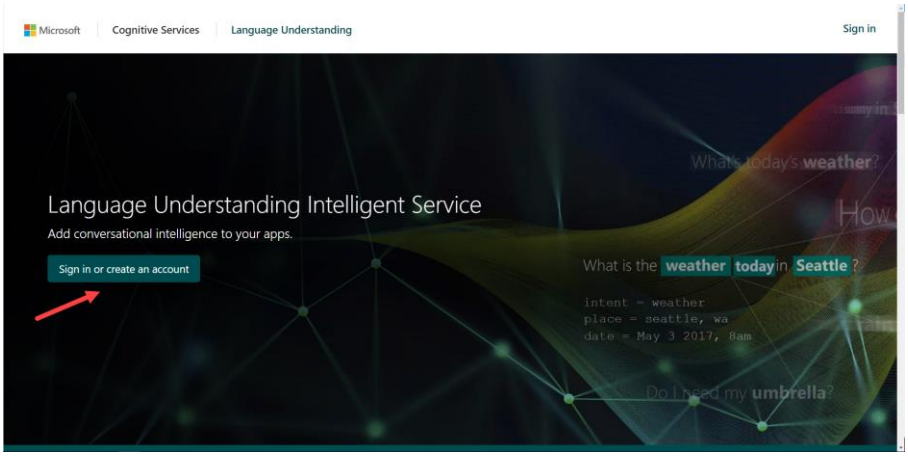
- If you are using the same browser that you used to create your Microsoft ID then you will be signed in automatically, otherwise you will need to use the ID you just created to sign in.
- Check the Terms of use box and click on Save.



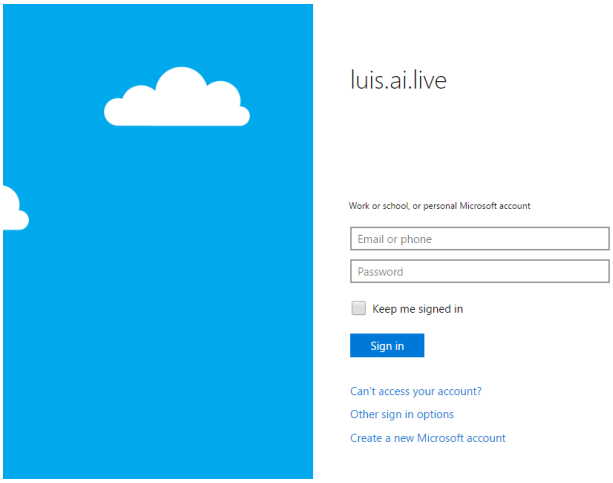


- You can leave this window open, we will be using it later.

3. Sign-up for LUIS. Language Understanding Intelligent Services
   - https://www.luis.ai/
   - Click on: Sign in or Create Account button

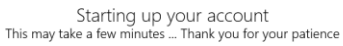


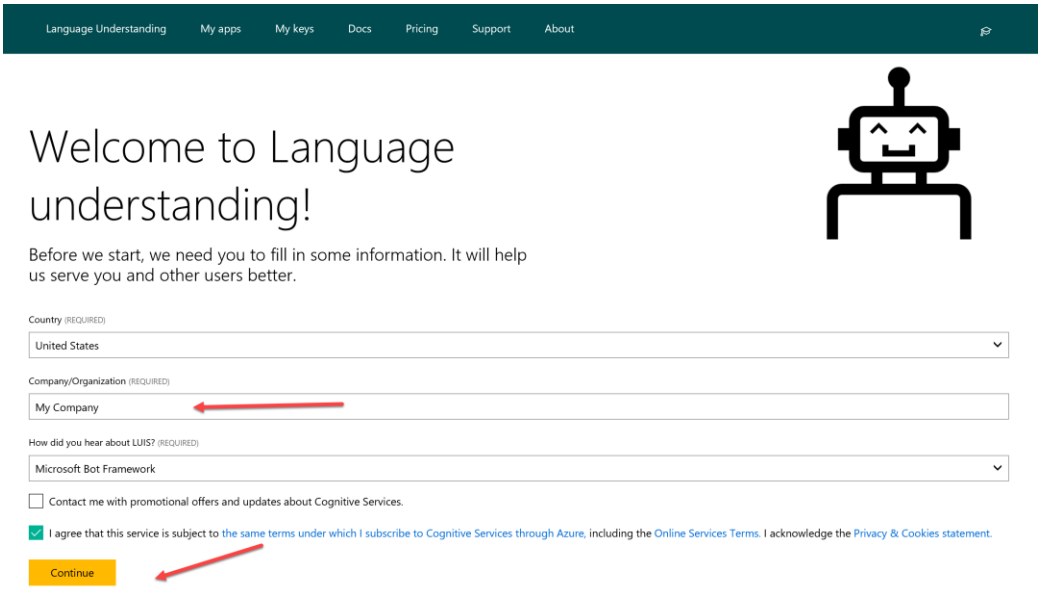
- Sign in with your Microsoft account



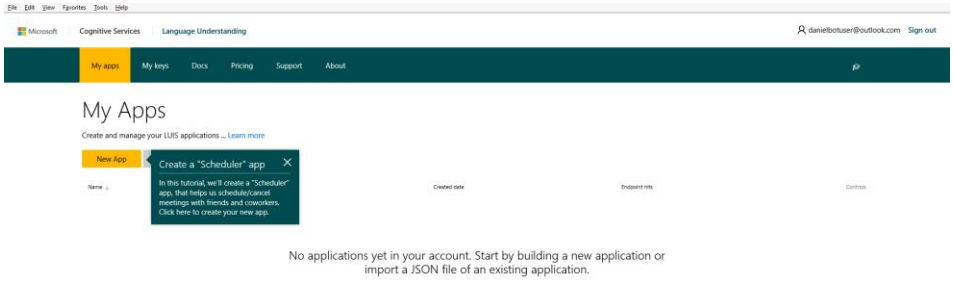
- If you are still signed in it will ask you to say Yes to accept permissions. Otherwise you will need to sign in with the Microsoft ID you created earlier.

- Fill out the required information (Put anything for company) and click Continue. (After it spins up)





That is all we need for now. We will come back to LUIS in another lab.



- We will explain and use this later for our bot.

Copy/Paste of Code

You will have the option to copy/paste code snippets from this document to complete this lab. You will learn much more by typing it in yourself but sometimes in a lab format speed is needed to get through all the exercises in time.

NOTE: If you are on a mac, you will be using the PDF file. Do not copy and paste from the PDF file. There is a separate file called SNIPSCSharp.txt that contain the snips you need.

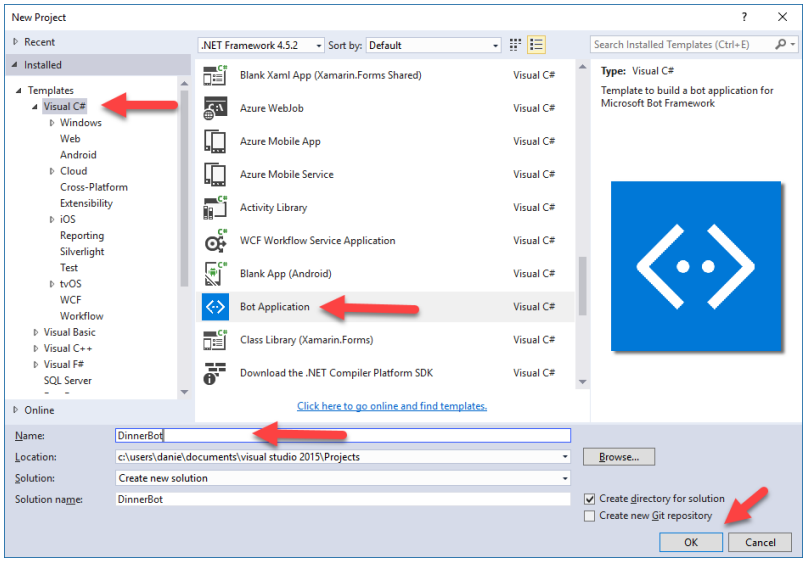
Exercise 1: Basic Bot using BotBuilder

In this exercise, you will create a simple bot using the bot framework C# teamplate and learn how rurn the emulator.

Detailed Steps

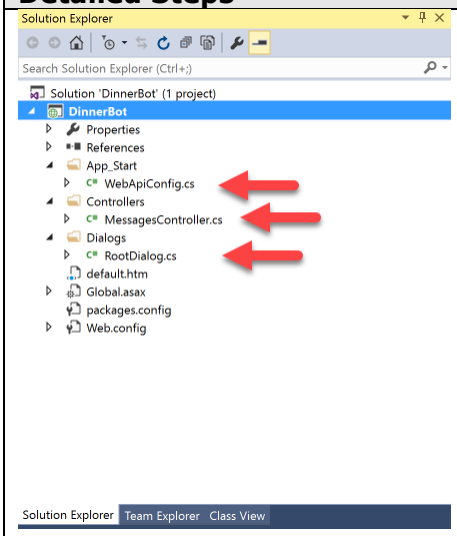
If you have not already done this in the prerequisites section, you will need to download and install the C# Bot Template. http://aka.ms/bf-bc-vstemplate (see instructions in Configuration and Setup section above)

1. Open or restart Visual Studio 2015 and go to **File → New → Project**
   Select the Bot Application Template and Name it DinnerBot



   If you have used Web API previously, you will notice that the project that was set up is very similar to a WebApi project.

Detailed Steps



You can see both a **MessagesController** (which we will look at in a second) and a **WebApiConfig** in addition to a **RootDialog**.. Let's open up the **WebApiConfig.cs**

```
public static void Register(HttpConfiguration config)
{
    // Json settings
    config.Formatters.JsonFormatter.SerializerSettings.NullValueHandling = NullValueHandling.Ignore;
    config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
    config.Formatters.JsonFormatter.SerializerSettings.Formatting = Formatting.Indented;
    JsonConvert.DefaultSettings = () => new JsonSerializerSettings()
    {
        ContractResolver = new CamelCasePropertyNamesContractResolver(),
        Formatting = Newtonsoft.Json.Formatting.Indented,
        NullValueHandling = NullValueHandling.Ignore,
    };

    // Web API configuration and services

    // Web API routes
    config.MapHttpAttributeRoutes();

    config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
```

In here, among other things, you can see our routes set up as api/{controller}/{id}. This is going to map to api/messages (The MessagesController). You will notice this route not just in your project but also when we set this up on the BotFramework Portal.

Now let's open up the **MessagesController.cs**

Detailed Steps

```
namespace DinnerBot
{
    [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)
        {
```

The first thing to notice is, as we discussed, it inherits from the ApiController . So any http Post to api/messages is routed to this method. Meaning all communication with your bot starts here. In addition, you can see it is being passed a type of Activity.

There are five different Activity Types.

ActivityType	Interface	Description
message	IMessageActivity	a simple communication between a user <-> bot
conversationUpdate	IConversationUpdateActivity	your bot was added to a conversation or other conversation metadata changed
contactRelationUpdate	IContactRelationUpdateActivity	The bot was added to or removed from a user's contact list
typing	ITypingActivity	The user or bot on the other end of the conversation is typing
ping	n/a	an activity sent to test the security of a bot.
deleteUserData	n/a	A user has requested for the bot to delete any profile / user data

NOTE: If your **MesasgeController.cs** file does not look like this and you don't have a RootDialog.cs file in the dialogs folder, you are using the OLD template. Delete it from %USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C# and follow the instructions in the Configuration and Setup section above.

In this template, the main activity, message is handled here in the post. While all others are handled in the HandleSystemMessage below.

```
[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;
    }
```

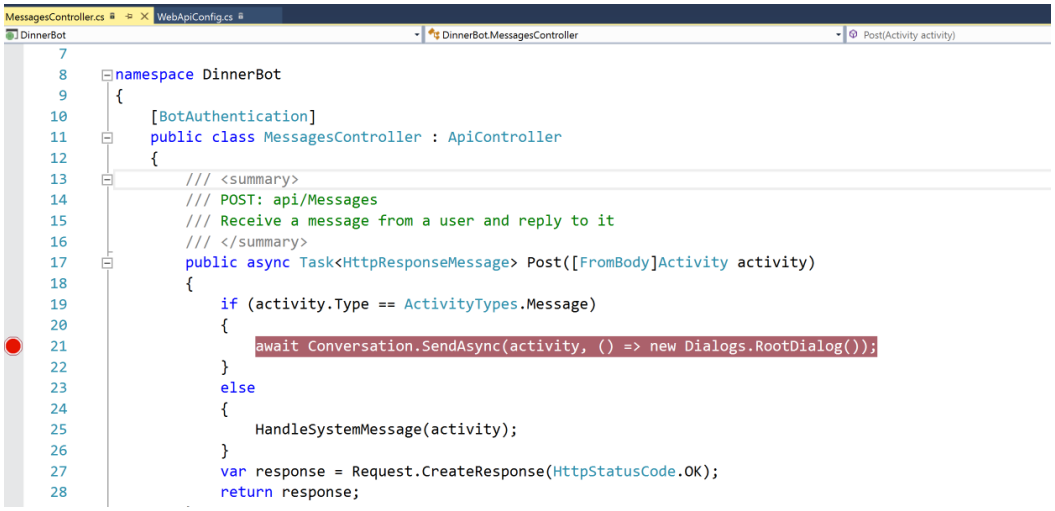
The **Post** message is marked with **asyc** because Bot Builder uses the C# facilities for handling asynchronous communication. So once we know it's a **Message**, we call **Conversation.SendAsync** and

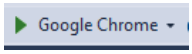
Detailed Steps

send the activity to a new **RootDialog**. The **RootDialog** will be the first stop for everything we will be doing in this bot.

We will be making changes to this bot but first we need to make sure that we can test it using the emulator. Make sure you have downloaded (https://docs.microsoft.com/en-us/bot-framework/resources-tools-downloads) and installed it before you begin.

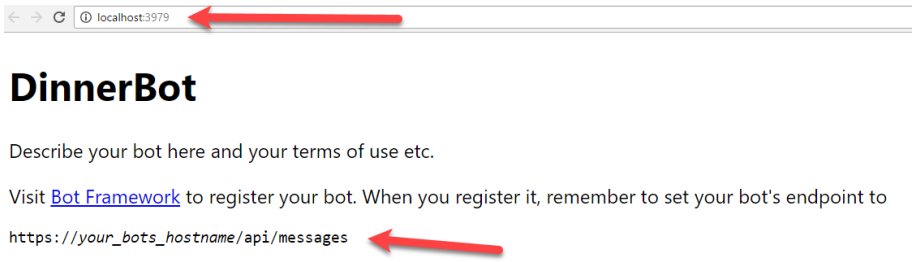
2. In Visual Studio, place a couple of breakpoints in the **MessagesController.cs** file so we can inspect things when we connect.



3. Hit **F5** or press the green arrow 

to run your project.

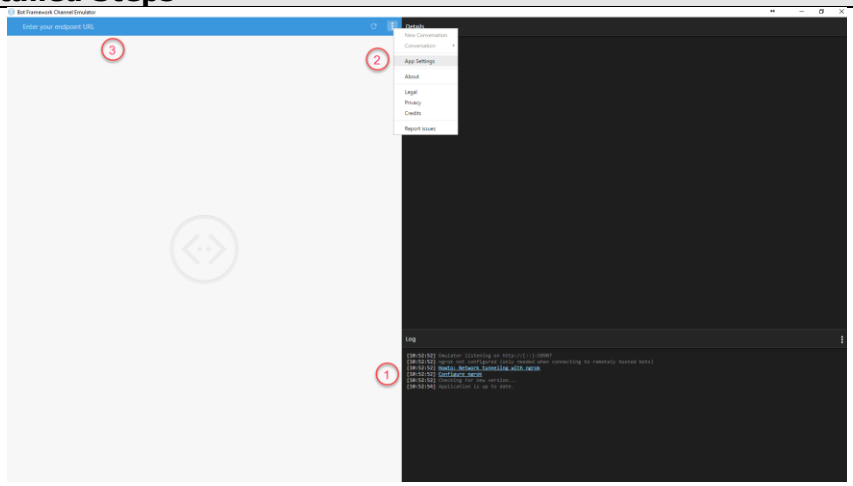
When it launches, you will see the following in your browser of choice.



Notice that the bot will launch on localhost:3979 and gives you a reminder of your bots endpoint as well. If you wanted you could use tool like **Paw**, **HTTPie**, or **Postman** to test our endpoint but instead we will use the emulator.

4. Run the Bot Framework Channel Emulator that you previously installed.

Detailed Steps

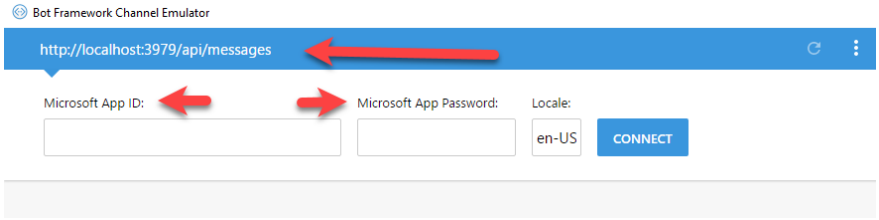


When it launches, you will notice a few things.
1) A log which shows the ServiceURL that the emulator is listening on, as well as a note to install NGrok which will be needed later for using the emulator with a cloud hosted bot.

2) An ellipse menu that can be used to set up NGrok, create conversations, and send messages.

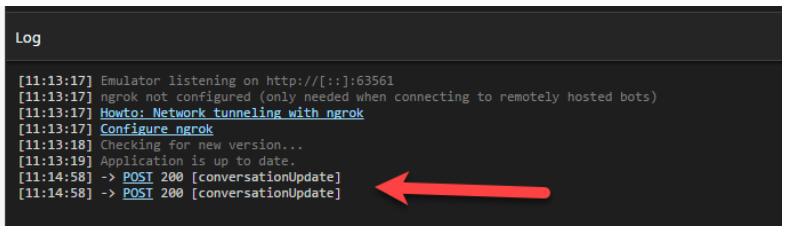
3) A prompt to enter the endpoint to your bot.

5. Click on the "Enter your endpoint URL" section to connect to your bot.
6. Enter the port that your bot launched on (Usually http://localhost:3979/api/messages)



notice that it is also asking for **Microsoft App ID** and **Microsoft App Password**. For testing locally, these are not needed.

7. Click on **CONNECT**. If all goes well, you should see 200 [ConversationUdate] twice in your log. Once for the user and once for the bot.



8. Next, type Hello (or anything you want) into the txt field of the emulator.

   Once you hit enter, you should hit the breakpoint you set in Visual Studio.

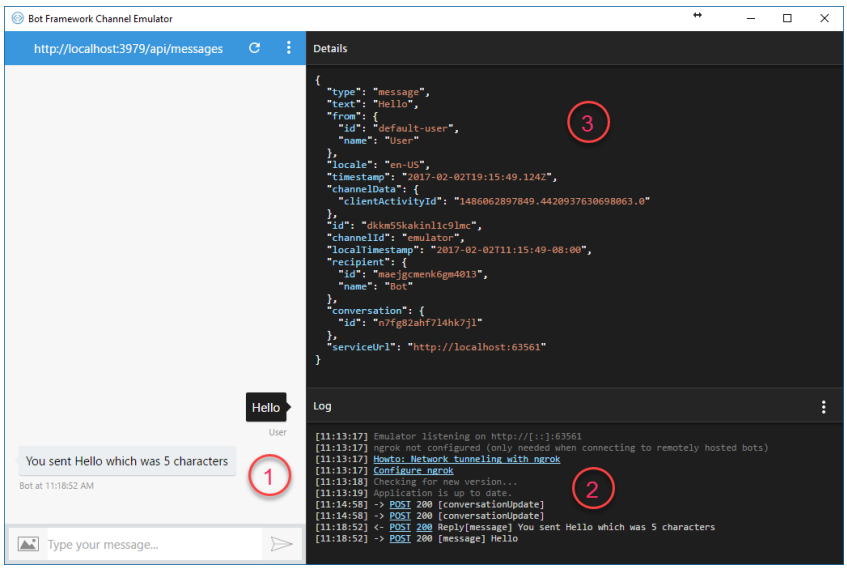
Detailed Steps

```
namespace DinnerBot
{
    [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;
        }
```

we are not going to walk through it, but take time to inspect the different values, properties and methods of the **Connector**, **Activity**, and **Message**.

When you are done, remove the breakponts and it **F5** to continure.

If you return back to the emulator, you will see the reponse from the bot (1), the entries in the log (2) and if you click on any of the post links, you will see the details associated with the request (3)



So in this section, we created a default hello world type of bot, got it up and running and interacted with it using the emulator. In the next section, we will start modifying it to create our dinner bot.

Exercise 2: Creating Dialogs

In this exercise, we will create a few simple dialogs in order to interact with the user.

Detailed Steps

The first dialog has already been created for you from the template. This is the RootDialog (found in the Dialogs Folder). This will be the place where all of our interaction flows.

Let's take a look at this file.

1. Double Click on RootDialog.cs to bring it up.

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

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

            return Task.CompletedTask;
        }
```

First notice that we mark the class as **[Serializable]**. The dialog stack and the state of all active dialogs are serialized to the per-user, per-conversation **IBotDataBag**. The serialized blob is persisted in the messages that the bot sends to and receives from the Connector. To be serialized, a Dialog class must include the [Serializable] attribute. All **IDialog** implementations in the Builder library are marked as serializable.

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

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

            return Task.CompletedTask;
        }
```

Next we implement the **IDialog<>** Interface. This interface has only one method **StartAsync** which is called when we create an instance of this dialog.

Detailed Steps

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

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

            return Task.CompletedTask;
        }
```

The **StartAsync** method calls **IDialogContext.Wait** with the continuation delegate to specify the method that should be called when a new message is received (**MessageReceivedAsync**). It is important to understand that the bot will wait here until the user sends a message. Then it will go to MessageReceivedAsync.

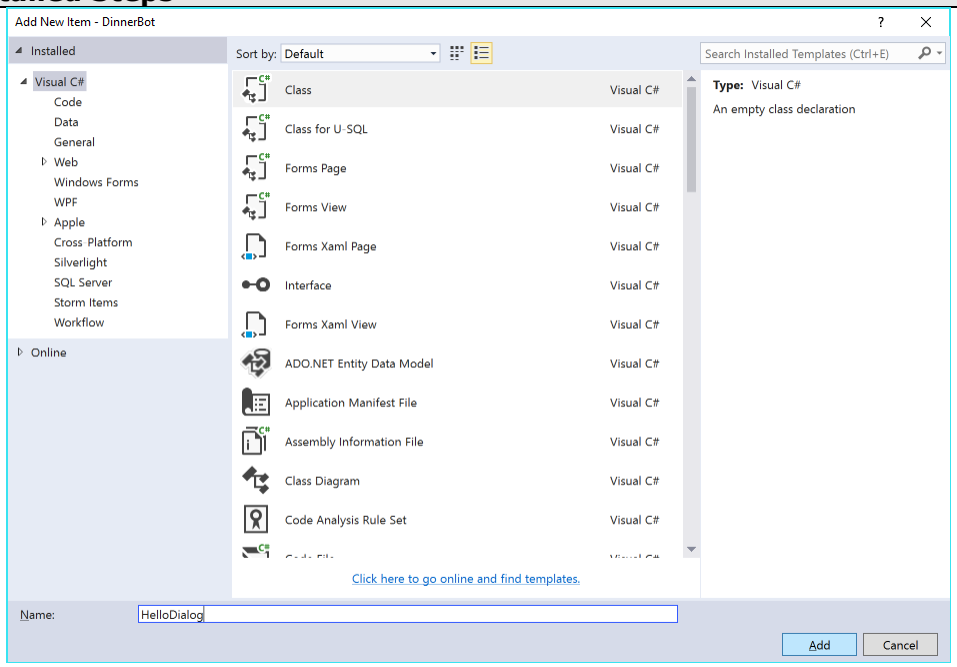
```
private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<object> result)
{
    var activity = await result as Activity;

    // 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);
}
```

In our sample we are simply just echoing back what the user said to the bot with the length of characters sent. We will be changing this. Keep in mind that the RootDialog.cs should function like more of a traffic cop, directing to the dialogs that will perform functions.

To do this, we will need to create another dialog.

2. Right click on the Dialogs Folder and select **Add → Class** and name it **HelloDialog.cs**.

Detailed Steps



Once this comes up, we need to add a few using statements for the Bot.

3. Add the following using statements to the top of the **HelloDialog.cs** file.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
```

----- **SNIP1**----------

```
using Microsoft.Bot.Builder.Dialogs;
using Microsoft.Bot.Connector;
```

Next, we need implement the **IDialog** Interface.

4. Add the **IDialog<object>** interface to the **HelloDialog** class and implement the interface.

Detailed Steps

This will create a method called **StartAsync** which is what is called when we call the dialog.

5. The Bot Framework requires that classes must be serialized so the bot can be stateless. So add the serializable attribute to the top of the class.

```
[Serializable]
public class HelloDialog : IDialog<object>
```

6. Replace the default **NotImplementedException** with the following.

```
namespace DinnerBot.Dialogs
{   [Serializable]
    public class HelloDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("Hi there, you made it to the Hello Dialog");
            context.Done<object>(null);
        }


    }
}
```

with the following code. Make sure you add the **async** keyword in front of Task in the method signature.

```
namespace DinnerBot.Dialogs
{   [Serializable]
    public class HelloDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("Hi there, you made it to the Hello Dialog");
            context.Done<object>(null);
        }


    }
}
```

When this dialog is called, it will post back the message to the user. And then will exit back to the RootDialog.

Detailed Steps

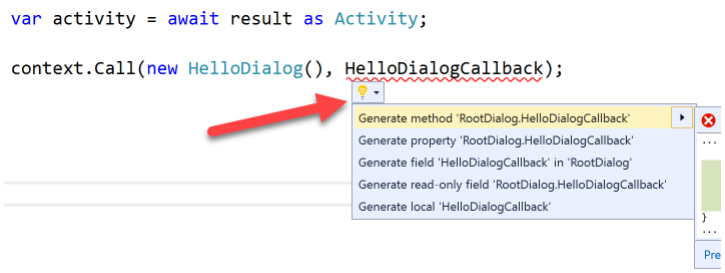
Now we need to make sure that this dialog is called from the **RootDialog**.

7. Open up the RootDialog.cs file and replace the code in the MessageReceivedAsync method with the following code .

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

    context.Call(new HelloDialog(), HelloDialogCallback);

}
```

We are using the context object to make a call out to the **HelloDialog**. We pass it the object (in this case a new **HelloDialog()** and a callback method for it to return to, called **HelloDialogCallback**. Let's implement that.

8. Hover over the **HelloDialogCallback** and select Generate method

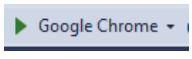


9. Replace the throw new NotImplementedException() with the following code and add the asyc classifier to the method.

```
private async Task HelloDialogCallback(IDialogContext context, IAwaitable<object> result)
{
    context.Wait(MessageReceivedAsync);
}
```

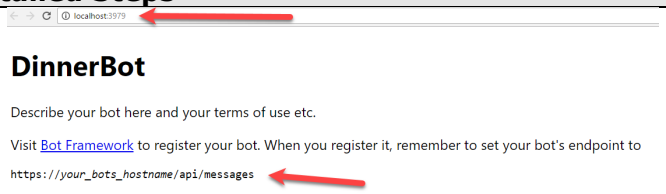
Since we are not passing anything back from the dialog at this point, all we want to do is have it wait for input and ready to go to the MessageReceivedAsync method. Of course, at this stage, all it will do is loop back to HelloDialog again since that is the only dialog we have.

Let's test our new dialog.

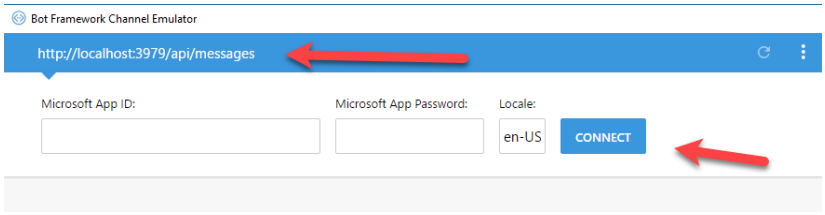
10. Hit **F5** or press the green arrow 

to run your project. Make sure the browser launches. (And remove the breakpoint in the MessageController if it is still there)

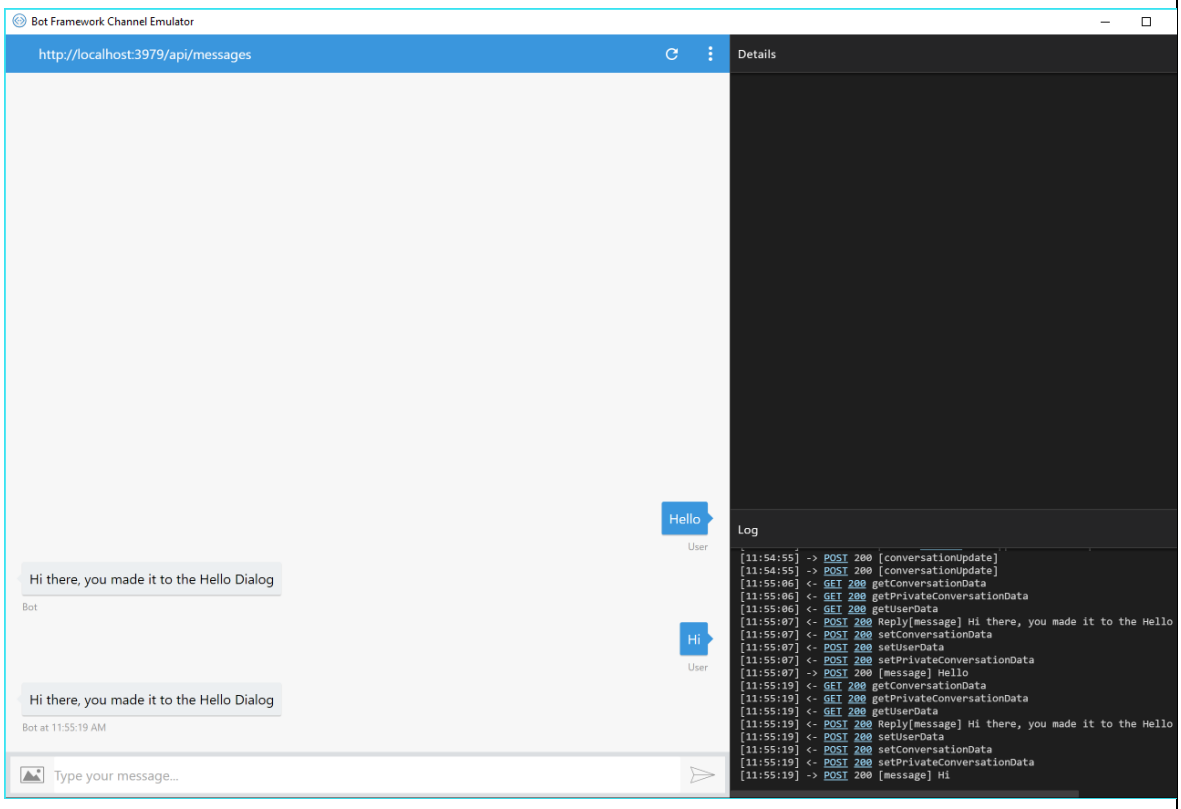
Detailed Steps



11. Open up the emulator and click on the top bar to revel the last connection we used and select connect.



Once the emulator launches, type in hello and the bot will now use our HelloDialog. No matter what you type it will go there and return to the root again.



Now that we have this working, lets make the HelloDialog actually do something other then sending a simple text message.

Detailed Steps

In the HelloDialog we are going to show how to save state to the state bag.

12. Inside you **HelloDialog.cs** file, place the following code inside the StartAsync method replacing what we have in there.

```
public async Task StartAsync(IDialogContext context)
{
    //Greet the user
    await context.PostAsync("Hey there, how are you?");
    //call the respond method below
    await Respond(context);
    //call context.Wait and set the callback method
    context.Wait(MessageReceivedAsync);
}
```

----- SNIP2----------

```
//Greet the user
await context.PostAsync("Hey there, how are you?");
//call the respond method below
await Respond(context);
//call context.Wait and set the callback method
context.Wait(MessageReceivedAsync);
```

Now we need to implement the **Respond** and **MessageReceivedAsync** methods. We pass the **context** into the respond method and use it to check state, and ask their name for later use.

13. Paste the following code **below** the **StartAsync** Method

```
private static async Task Respond(IDialogContext context)
{
    //Varible to hold user name
    var userName = String.Empty;
    //check to see if we already have username stored
    context.UserData.TryGetValue<string>("Name", out userName);
    //If not, we will ask for it.
    if (string.IsNullOrEmpty(userName))
    {
        //We ask here but dont capture it here, we do that in the MessageRecieved Async
        await context.PostAsync("What is your name?");
        //We set a value telling us that we need to get the name out of userdata
        context.UserData.SetValue<bool>("GetName", true);
    }
    else
    {
        //If name was already stored we will say hi to the user.
        await context.PostAsync(String.Format("Hi {0}.  How can I help you today?", userName));
    }
```

----- SNIP3----------

```
private static async Task Respond(IDialogContext context)
 {
     //Variable to hold user name
     var userName = String.Empty;
     //check to see if we already have username stored
     context.UserData.TryGetValue<string>("Name", out userName);
```

Detailed Steps

```
     //If not, we will ask for it.
     if (string.IsNullOrEmpty(userName))
     {
         //We ask here but dont capture it here, we do that in the
MessageRecieved Async
         await context.PostAsync("What is your name?");
         //We set a value telling us that we need to get the name out
of userdata
         context.UserData.SetValue<bool>("GetName", true);
     }
     else
     {
         //If name was already stored we will say hi to the user.
         await context.PostAsync(String.Format("Hi {0}.  How can I help
you today?", userName));
     }
 }
```

14. Now post the following code **below** the **Respond** method. In here we use the IMessageActivity that is passed in to capture what the user typed when we asked their name.

```
public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)
{
    //variable to hold message coming in
    var message = await argument;
    //variable for userName
    var userName = String.Empty;
    //variable to hold whether or not we need to get name
    var getName = false;
    //see if name exists
    context.UserData.TryGetValue<string>("Name", out userName);
    //if GetName exists we assign it to the getName variable and replace false
    context.UserData.TryGetValue<bool>("GetName", out getName);
    //If we need to get name, we go in here.
    if (getName)
    {
        //we get the username we stored above. and set getname to false
        userName = message.Text;
        context.UserData.SetValue<string>("Name", userName);
        context.UserData.SetValue<bool>("GetName", false);
    }

    //we call respond again, this time it will print out the name and greeting
    await Respond(context);
    //call context.done to exit this dialog and go back to the root dialog
    context.Done(message);
}
```

----- SNIP4----------

```
public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> argument)
{
    //variable to hold message coming in
    var message = await argument;
    //variable for userName
    var userName = String.Empty;
```

Detailed Steps

```
    //variable to hold whether or not we need to get name
    var getName = false;
    //see if name exists
    context.UserData.TryGetValue<string>("Name", out userName);
    //if GetName exists we assign it to the getName variable and
replace false
    context.UserData.TryGetValue<bool>("GetName", out getName);
    //If we need to get name, we go in here.
    if (getName)
    {
        //we get the username we stored above. and set getname to false
        userName = message.Text;
        context.UserData.SetValue<string>("Name", userName);
        context.UserData.SetValue<bool>("GetName", false);
    }

    //we call respond again, this time it will print out the name and
greeting
    await Respond(context);
    //call context.done to exit this dialog and go back to the root
dialog
    context.Done(message);
}
```

The code is well commented, take your time to see how things are used in the dialog.

Now we want to wire up the **RootDialog** a little better in order to send the user into the **HelloDialog** and receive back data**.**

15. Open up the **RootDialog.cs** file and add two strings to the top of the class to represent the choices.

```
[Serializable]
public class RootDialog : IDialog<object>
{
    private const string ReservationOption = "Reserve Table";
    private const string HelloOption       = "Say Hello"    ;
```

----- SNIP5----------

```
private const string ReservationOption = "Reserve Table";
private const string HelloOption = "Say Hello";
```

Detailed Steps

Now we want to use one of the built-in Dialogs. We will use the PromptDialog.Choice dialog to give them an option. We are going to prompt them right after they are greeted when they start a conversation.

16. Paste the following code inside the **MessageReceivedAsync** method in the **RootDialog.cs** file. (REPLACING WHAT IS IN THERE)
This will let them choose between reserving a table or just saying hello.

```
private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
{
    PromptDialog.Choice(
        context,
        this.OnOptionSelected,
        new List<string>() { ReservartionOption, HelloOption },
        String.Format("Hi, are you looking for to resere a table or Just say hello?"), "Not a valid option", 3);
}
```

----- SNIP6----------

```
PromptDialog.Choice(
     context,
     this.OnOptionSelected,
     new List<string>() { ReservationOption, HelloOption },
     String.Format("Hi, are you looking for to reserve a table or Just
say hello?"), "Not a valid option", 3);
```

This code passes in the context, sets a callback method (OnOptionSelected), defines a message when an invalid option is selected and limits try's to 3. We will handle the try limit in the call back function. Let's implement that now.

17. Since we are using a list, add the System.Collections.Generic using statement to the top of the file.

```
using System.Collections.Generic;
```

18. In the **RootDialog.cs** file place the following code below the **MessageReceivedAsync** method.

Detailed Steps

```
private async Task OnOptionSelected(IDialogContext context, IAwaitable<string> result)
{
    try
    {
        //capture which option then selected
        string optionSelected = await result;
        switch (optionSelected)
        {
            case ReservartionOption:
                break;

            case HelloOption:
                context.Call(new HelloDialog(), this.ResumeAfterOptionDialog);
                break;
        }
    }
    catch (TooManyAttemptsException ex)
    {
        //If too many attempts we send error to user and start all over.
        await context.PostAsync($"Ooops! Too many attemps :( You can start again!");

        //This sets us in a waiting state, after running the prompt again.
        context.Wait(this.MessageReceivedAsync);
    }
}
```

----- SNIP7----------

```
private async Task OnOptionSelected(IDialogContext context,
IAwaitable<string> result)
{
    try
    {
        //capture which option then selected
        string optionSelected = await result;
        switch (optionSelected)
        {
            case ReservationOption:
                break;

            case HelloOption:
                context.Call(new HelloDialog(),
this.ResumeAfterOptionDialog);
                break;
        }
    }
    catch (TooManyAttemptsException ex)
    {
        //If too many attempts we send error to user and start all
over.
        await context.PostAsync($"Ooops! Too many attempts :( You can
start again!");

        //This sets us in a waiting state, after running the prompt
again.
        context.Wait(this.MessageReceivedAsync);
```

Detailed Steps

```
        }
    }
```

There are a couple of important parts of this code. If they selected the HelloOption then they will be sent to the **HelloDialog** by using **context.call**.

```
case HelloOption:
    context.Call(new HelloDialog(), this.ResumeAfterOptionDialog);
    break;
```

when it finishes that dialog it will return to the **ResumeAfterOptionsDialog** method as show in the code above so we will need to implement that method.

19. Paste the following code below the **OnOptionSelected** method in the **RootDialog.cs** file. In this code we are retrieving the message back from the Dialog (but doing nothing with it), capturing any errors coming back, and setting it ready for the user to communicate again with the call to context.wait.

```
private async Task ResumeAfterOptionDialog(IDialogContext context, IAwaitable<object> result)
{
    try
    {
        var message = await result;
    }
    catch (Exception ex)
    {
        await context.PostAsync($"Failed with message: {ex.Message}");
    }
    finally
    {
        context.Wait(this.MessageReceivedAsync);
    }
}
```

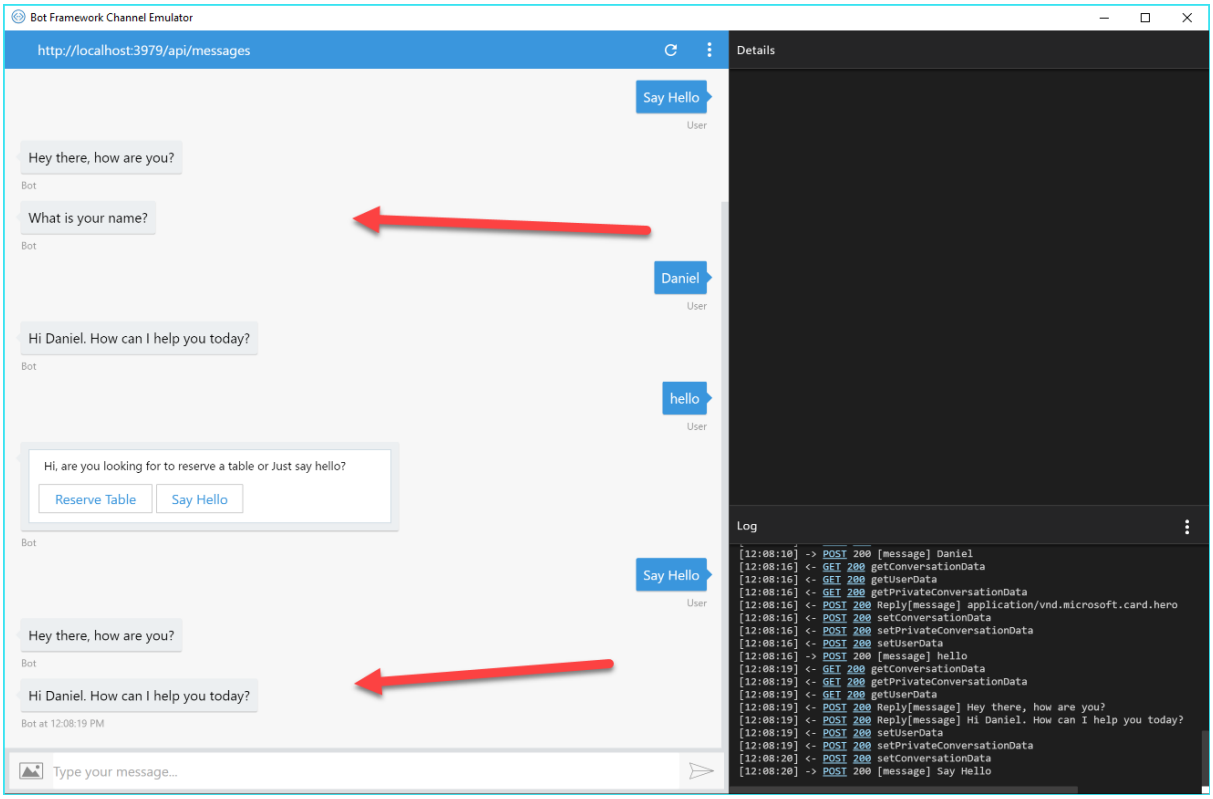
----- SNIP8----------

```
private async Task ResumeAfterOptionDialog(IDialogContext context,
IAwaitable<object> result)
{
    try
    {
        var message = await result;
    }
    catch (Exception ex)
    {
        await context.PostAsync($"Failed with message: {ex.Message}");
    }
    finally
    {
        context.Wait(this.MessageReceivedAsync);
```

Detailed Steps

```
        }
}
```

Run your project and connect it to the emulator to test. (Detailed instructions if needed above) .



You will notice the second time that I say hello. It does not ask for my name, but pulls it out of UserData.

If you look at the code in the **HelloDialog** you can see the potential for unintended use, meaning we are not checking values, of confirming, or validating data. We could of course write all that by hand but we don't need to. In the next exercise, we will use FormFlow to help us with this.

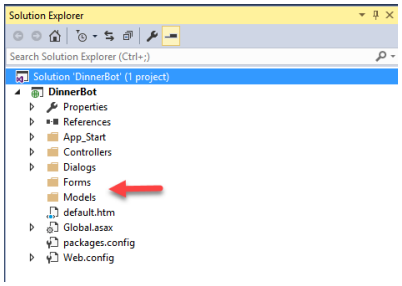
Exercise 3: Form Flow

In this exercise, we will be using FormFlow to create a dialog. There are a few ways to implement FormFlow, we will utilize prompts.

Detailed Steps

As we continue to work on the DinnerBot project, we will be enhancing the project to incorporate the different ways to build a bot. One of those, in the C# SDK, is the use of **FormFlow**. There are a few different ways to create FormFlows. We will utilize the separation of the model that the form flow follows, and the form itself. So to start we will need to create a couple of new folders.

1. Open up the DinnerBot project in Visual Studio and in the Solution Explorer, right click on the DinnerBot project and create two new folders called **Forms** and **Models**



2. Next right click on the **Models** Folder and create a class called **Reservation.cs**.
3. Add the **[Serializable]** attribute to the top of the class.
4. Add the following Using Statements to the top of the class.

using Microsoft.Bot.Builder.FormFlow;

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Web;
5   using Microsoft.Bot.Builder.FormFlow;
6
7   namespace DinnerBot.Models
8   {
9       [Serializable]
0       public class Reservation
1       {
2
3       }
4   }
```

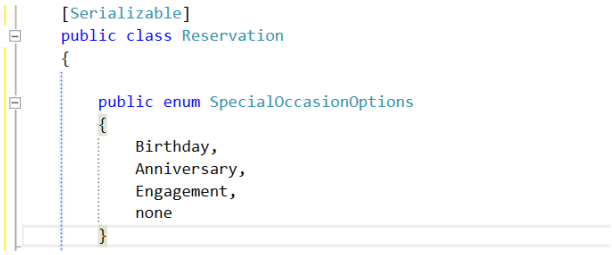
You will notice that we do not need to implement the IDialog Interface for this class. FormFlow will take care of that for us.

We will be utilizing a few different techniques for things like validation to show the multiple ways of doing them and to show how flexible FormFlow is. We are essentially creating a class, with properties and methods, that FormFlow will use to create a conversation for us. In this

Detailed Steps

case, it is for a reservation for a restaurant. Let's get started by making some properties.

5. The first thing we need is to create an Enum to provide the ability for one of the answers from the questions to come from a list. Inside the class, paste the following code for Special Occasion selection.



```
[Serializable]
public class Reservation
{

    public enum SpecialOccasionOptions
    {
        Birthday,
        Anniversary,
        Engagement,
        none
    }
```

----- SNIP9----------

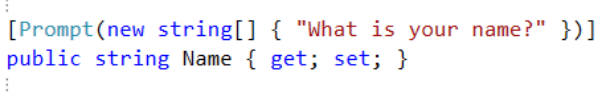
```
public enum SpecialOccasionOptions
{
   Birthday,
   Anniversary,
   Engagement,
   none
}
```

6. Next, we need to add a couple of properties for data we would like to collect from the user. Add the following properties below the enum.

----- SNIP10----------

```
[Prompt(new string[] { "What is your name?" })]
public string Name { get; set; }

[Prompt(new string[] { "What is your email?" })]
public string Email { get; set; }

[Pattern(@"^(\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$")]
public string PhoneNumber { get; set; }
```

Let's look at these individually. The first one is a simple string with a [Prompt] attribute that sets the question FormFlow will ask the user.



```
[Prompt(new string[] { "What is your name?" })]
public string Name { get; set; }
```

The second one is also a string to collect the email

Detailed Steps

```
[Prompt(new string[] { "What is your email?" })]
public string Email { get; set; }
```

The third one is a bit different, it uses a [Pattern] attribute to validate the phone number using a regular expression. We could have done that for the email as well but we will do that differently later on.

```
[Pattern(@"^(\+\d{1,2}\s)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$")]
public string PhoneNumber { get; set; }
```

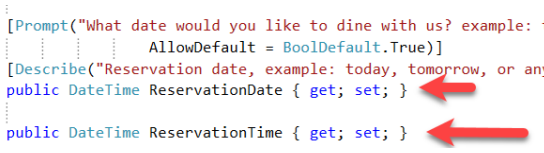
7. The next two properties will be for Reservation Date and Reservation Time. Paste them below the PhoneNumber property

----- **SNIP11**----------

```
[Prompt("What date would you like to dine with us? example: today, tomorrow, or any date like 04-06-2017 {||}", AllowDefault = BoolDefault.True)]
[Describe("Reservation date, example: today, tomorrow, or any date like 04-06-2017")]
public DateTime ReservationDate { get; set; }

public DateTime ReservationTime { get; set; }
```

ReservationDate not only utilizes a **[Prompt]** attribute, but also a **[Describe]** attribute, which will be shown to the user if they type help during this FormFlow

ReservationTime on the other hand is just a property. It will still be validated to make sure that they give an answer that formats to a **DateTime**. That is part of the magic of FormFlow.



8. The final two properties are for **NumberOfDinners**, **SpecialOccasionOptions** (using the Enum) and Ratings to show that some can be optional. Paste the following code under the **ReservationTime** property.

----- **SNIP12**----------

```
[Prompt("How many people will be joining us?")]
[Numeric(1, 20)]
public int? NumberOfDinners;
```

Detailed Steps

```
public SpecialOccasionOptions? SpecialOccasion;

[Numeric(1, 5)]
[Optional]
[Describe("for how you enjoyed your experience with Dinner Bot today
(optional)")]
public double? Rating;
```

9. The last thing we want to add to this class is a constructor. Inside FormFlow you will not automatically have access to your current context or to data held in your userData. In our instance, we are already asking the user for their name, so we don't want to ask them for it again when they are creating a reservation. You could easily pass in the entire context in, but we only need name so we pass it in the constructor and set the Name property to what is passed in.

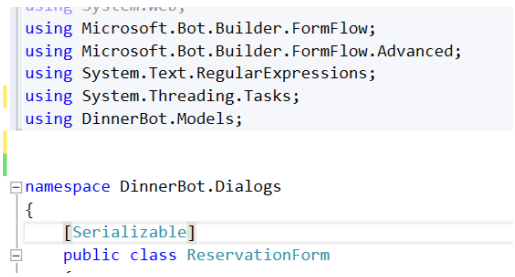
   Past the following code at the top of the class above the enum.

   ----- SNIP13----------

```
public Reservation(string name )
        {
            this.Name = name;
        }
```

10. Now we need to create the build form. Right click on the **Form** folder and create a class called **ReservationForm.cs**
11. Add the **[Serializable]** attribute to the top of the class.
12. Add the following Using Statements to the top of the class.

    using Microsoft.Bot.Builder.FormFlow;
    using Microsoft.Bot.Builder.FormFlow.Advanced;
    using System.Text.RegularExpressions;
    using System.Threading.Tasks;
    using DinnerBot.Models;



Detailed Steps

13. Inside the class, paste the following code.

----- SNIP14----------

```
public static IForm<Reservation> BuildForm()
{
    return new FormBuilder<Reservation>()
        .Field(nameof(Reservation.Name))
        .Field(nameof(Reservation.Email), validate:
ValidateContactInformation)
        .Field(nameof(Reservation.PhoneNumber))
        .Field(nameof(Reservation.ReservationDate))
        .Field(new
FieldReflector<Reservation>(nameof(Reservation.ReservationTime))
            .SetPrompt(PerLinePromptAttribute("What time would you like
to arrive?"))
            ).AddRemainingFields()
        .Build();
}
```

We use the **IForm** of type **Reservation** to return a **FormBuilder**(of the same type).
We set the order for the first few fields, as you can see, we use a custom validator for the email as opposed to using the pattern like we did for phone. This gives us more flexibility. We can also set the prompt type per as you can see for the **ReservationTime** field. We then call **AddRemainingFields()** to pull in the rest. They will be pulled in the order they show up in the model. Finally, we call build.

```
public static IForm<Reservation> BuildForm()
{
    return new FormBuilder<Reservation>()
        .Field(nameof(Reservation.Name))
        .Field(nameof(Reservation.Email), validate: ValidateContactInformation)
        .Field(nameof(Reservation.PhoneNumber))
        .Field(nameof(Reservation.ReservationDate))
        .Field(new FieldReflector<Reservation>(nameof(Reservation.ReservationTime))
            .SetPrompt(PerLinePromptAttribute("What time would you like to arrive?"))
            ).AddRemainingFields()
        .Build();
}
```

14. Next, we add the validation code that we are using in the build. Paste the following code underneath the BuildForm() method. We won't examine this since it is basic validation code.

----- SNIP15----------

```
private static Task<ValidateResult>
ValidateContactInformation(Reservation state, object response)
{
    var result = new ValidateResult();
```

Detailed Steps

```
    string contactInfo = string.Empty;
    if (GetEmailAddress((string)response, out contactInfo))
    {
        result.IsValid = true;
        result.Value = contactInfo;
    }
    else
    {
        result.IsValid = false;
        result.Feedback = "You did not enter valid email address.";
    }
    return Task.FromResult(result);
}

private static bool GetEmailAddress(string response, out string
contactInfo)
{
    contactInfo = string.Empty;
    var match = Regex.Match(response, @"[a-z0-9!#$%&'*+/=?^_`{|}~-
]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@(?:[a-z0-9](?:[a-z0-9-]*[a-z0-
9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?");
    if (match.Success)
    {
        contactInfo = match.Value;
        return true;
    }
    return false;
}

private static PromptAttribute PerLinePromptAttribute(string pattern)
{
    return new PromptAttribute(pattern)
    {
        ChoiceStyle = ChoiceStyleOptions.PerLine
    };
}
```

15. Now before we wire this up, we want to clean a few things up. The **HelloDialog** is doing more than just saying hello, it is also asking for a name and saving it. We want to abstract that out to its own dialog to hold User Info. Right-click on the Dialogs folder and **Add → Class** and call it **UserInfoDialog.cs**

Making sure to:
Add the following using statements
using Microsoft.Bot.Builder.Dialogs ;
using Microsoft.Bot.Connector ;

Implement the **IDialog<IMessageActivity>** interface,

Detailed Steps

Make the class **[Serializable]**
Add the **async** qualifier to the **StartAsync** method
(We will be pasting in the rest)
(For detailed instructions refer back to creating the **HelloDialog** *above)*

SPECIAL NOTE: Make sure the IDialog<> interface is using IMessageActivity and not Object!! We will be passing back data to the callback method this time.

16. In the StartAsync method paste the following code. Replacing the **throw new NotImplementedException();**

----- SNIP16----------

```
//Greet the user
await context.PostAsync("Before we begin, we would like to know who we
are talking to?");
//call the respond method below
await Respond(context);
//call context.Wait and set the callback method
context.Wait(MessageReceivedAsync);
```

17. Next, we want to implement the **Respond()** method. Paste the following below the **StartAsync** method.

----- SNIP17----------

```
private static async Task Respond(IDialogContext context)
{
    //Variable to hold user name
    var userName = String.Empty;
    //check to see if we already have username stored
    context.UserData.TryGetValue<string>("Name", out userName);
    //If not, we will ask for it.
    if (string.IsNullOrEmpty(userName))
    {
        //We ask here but dont capture it here, we do that in the
MessageRecieved Async
        await context.PostAsync("What is your name?");
        //We set a value telling us that we need to get the name out of
userdata
        context.UserData.SetValue<bool>("GetName", true);
    }
```

Detailed Steps

```
    else
    {
        //If name was already stored we will say hi to the user.
        await context.PostAsync(String.Format("Hi {0}.  How can I help
you today?", userName));

    }
}
```

18. Now to complete the dialog, add the following **MessageReceivedAsync** method below StartAsync method.

----- SNIP18----------

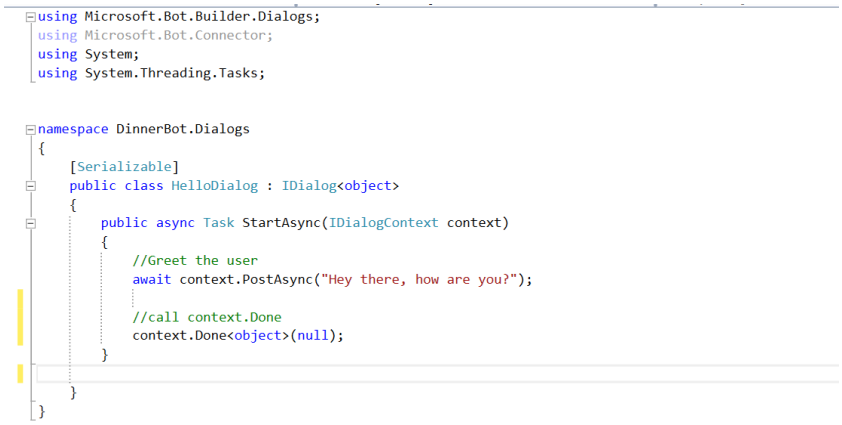
```
public async Task MessageReceivedAsync(IDialogContext context,
IAwaitable<IMessageActivity> argument)
{
    //variable to hold message coming in
    try
    {
        var message = await argument;
        //variable for userName
        var userName = String.Empty;
        //variable to hold whether or not we need to get name
        var getName = false;
        //see if name exists
        context.UserData.TryGetValue<string>("Name", out userName);
        //if GetName exists we assign it to the getName variable and
replace false
        context.UserData.TryGetValue<bool>("GetName", out getName);
        //If we need to get name, we go in here.
        if (getName)
        {
            //we get the username we stored above. and set getname to
false
            userName = message.Text;
            context.UserData.SetValue<string>("Name", userName);
            context.UserData.SetValue<bool>("GetName", true);

            context.Wait(MessageReceivedAsync);
        }
        //await Respond(context);
        context.Done(message);
    }
```

Detailed Steps

```
    catch (Exception ex)
    {

        string message = ex.Message;
    }


}
```

Since we have already seen similar code in the **HelloDialog** we will not discuss it again here.

And speaking of the **HelloDialog**, we need to trim that a bit. Since we are gathering the name in the **UserInfoDialog**, all we need here is to say hi. Remove all except the following



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


namespace DinnerBot.Dialogs
{
    [Serializable]
    public class HelloDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            //Greet the user
            await context.PostAsync("Hey there, how are you?");

            //call context.Done
            context.Done<object>(null);
        }

    }
}
```

We should be left with just two lines in the StartAsync as shown above. If you would like to just replace the contents of the class file, you can use the snip below.
----- SNIP19----------

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


namespace DinnerBot.Dialogs
{
    [Serializable]
    public class HelloDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            //Greet the user
```

Detailed Steps

```
            await context.PostAsync("Hey there, how are you?");

            //call context.Done
            context.Done<object>(null);
        }

    }
}
```

Now we want to go back to our Root Dialog and make some changes in order to call both our hello and our reservation dialogs. We want to set up some simple logic to check and see if we already know the name of the user and if not, call the **UserInfoDialog**.

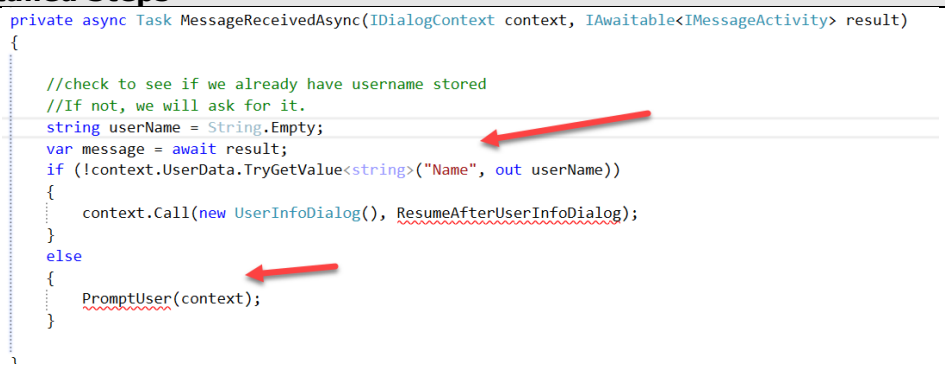
19. Open up **RootDialog.cs** and go to the **MessageReceivedAsync** method. Add the following code (Replacing what is currently there)

----- SNIP20----------

```
//check to see if we already have username stored
//If not, we will ask for it.
string userName = String.Empty;
var message = await result;
if (!context.UserData.TryGetValue<string>("Name", out userName))
{
    context.Call(new UserInfoDialog(), ResumeAfterUserInfoDialog);
}
else
{
    PromptUser(context);
}
```

In the code we are first checking to see if Name is already stored in **UserData**, if not we use **context.Call** to go into the UserInfoDialog and get the users name. Once we have the name we go back to prompt the user. Since we will be calling this from a few places we have abstracted that out to its own method called **PromptUser** so we need to implement that.

Detailed Steps



```
private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
{

    //check to see if we already have username stored
    //If not, we will ask for it.
    string userName = String.Empty;
    var message = await result;
    if (!context.UserData.TryGetValue<string>("Name", out userName))
    {
        context.Call(new UserInfoDialog(), ResumeAfterUserInfoDialog);
    }
    else
    {
        PromptUser(context);
    }

}
```

20. Right under the **StartAsync** method, add the following code.

----- SNIP21----------

```
private void PromptUser(IDialogContext context)
{
    PromptDialog.Choice(
    context,
    this.OnOptionSelected,
    // Present two (2) options to user
    new List<string>() { ReservationOption, HelloOption },
    String.Format("Hi {0}, are you looking for to reserve a table or
Just say hello?", context.UserData.Get<String>("Name")), "Not a valid
option", 3);
}
```

This now interjects the name we saved into the prompt since we will always be asking the name first. We do that by having the **StartAsync** method always call the **MessageReceivedAsync** method with a **context.Wait()**.

21. The last thing we need to do for this section is to implement the **ResumeAfterUserInfoDialog**. Paste the following code below the **MessageReceivedAsync** Method

----- SNIP22----------

```
private async Task ResumeAfterUserInfoDialog(IDialogContext context,
IAwaitable<object> result)
{
    PromptUser(context);
}
```

This will just call our PromptUser once it returns.

Detailed Steps

Now we want to update our **optionSelected** case statement inside of our **OnOptionSelected** method with the call to our **ReservationDialog.** We call this slightly differently since we are using Form Flow. In the context.Call, we pass it the Reservation with the name collected and saved in userData. Since we already asked them, we don't want to ask again for reservations. We then call the BuildForm method of that dialog, and finally give it a call back method (which we will create shortly).

22. Paste the following code inside switch statement in the OnOptionsSelected method. This not only includes the new code we need to create the reservation form, but also a new callback method for the HelloOption which we will create next.

----- SNIP23----------

```
case ReservationOption:

    var form = new FormDialog<Reservation>(
    new Reservation(context.UserData.Get<String>("Name")),
    ReservationForm.BuildForm,
    FormOptions.PromptInStart,
    null);

    context.Call(form, this.ReservationFormComplete);
    break;

case HelloOption:
    context.Call(new HelloDialog(), this.ResumeAfterUserHelloDialog);
    break;
```

Detailed Steps

You will need to add the following using statements to the top of your file.
using DinnerBot.Models;
using DinnerBot.Forms;
using Microsoft.Bot.Builder.FormFlow;

We are almost there, we need to create two callback methods. One simple one for the new HelloDialog Callback and one for the Reservation Form callback. This is where we can see the results generated by the FormFlow.

23. First, we will create the method for the **HelloDialog** callback. This is going to be exactly the same as the callback for the **ResumeAfterUserInfoDialog**. Paste the following code above the **MessageReceivedAsync** Method.

----- SNIP24----------

```
private async Task ResumeAfterUserHelloDialog(IDialogContext context,
IAwaitable<object> result)
{
    //we want it to go right to the prompting of reservation or hello
    PromptUser(context);
}
```

24. Next paste the following code below the StartAsync method. It is a lot of code but we will walk through it after pasting.

----- SNIP25----------

```
private async Task ReservationFormComplete(IDialogContext context,
IAwaitable<Reservation> result)
{
    try
    {
        var reservation = await result;
        await context.PostAsync("Thanks for the using Dinner Bot.");
        //use a card for showing their data
        var resultMessage = context.MakeMessage();
        //resultMessage.AttachmentLayout =
AttachmentLayoutTypes.Carousel;
        resultMessage.Attachments = new List<Attachment>();
        string ThankYouMessage;

        if (reservation.SpecialOccasion ==
Reservation.SpecialOccasionOptions.none)
        {
            ThankYouMessage = reservation.Name + ", thank you for
joining us for dinner, we look forward to having you and your guests.";
```

Detailed Steps

```
        }
        else
        {
            ThankYouMessage = reservation.Name + ", thank you for
joining us for dinner, we look forward to having you and your guests
for the " + reservation.SpecialOccasion;
        }
        ThumbnailCard thumbnailCard = new ThumbnailCard()
        {

            Title = String.Format("Dinner Reservations on {0}",
reservation.ReservationDate.ToString("MM/dd/yyyy")),
            Subtitle = String.Format("at {1} for {0} people",
reservation.NumberOfDinners,
reservation.ReservationTime.ToString("hh:mm")),
            Text = ThankYouMessage,
            Images = new List<CardImage>()
        {
            new CardImage() { Url =
"https://upload.wikimedia.org/wikipedia/en/e/ee/Unknown-person.gif" }
        },
        };

        resultMessage.Attachments.Add(thumbnailCard.ToAttachment());
        await context.PostAsync(resultMessage);
        await context.PostAsync(String.Format(""));
    }
    catch (FormCanceledException)
    {
        await context.PostAsync("You canceled the transaction, ok. ");
    }
    catch (Exception ex)
    {
        var exDetail = ex;
        await context.PostAsync("Something really bad happened. You can
try again later meanwhile I'll check what went wrong.");
    }
    finally
    {
        context.Wait(MessageReceivedAsync);
    }
}
```

We will start at the beginning of the method.
The **reservation** variable will hold the result of the form. After a quick prompt to the user, we create variables for the result message (we will use this to present a thumbnail card) and a variable for a thank you message.

Detailed Steps

```
var reservation = await result;
await context.PostAsync("Thanks for the using Dinner Bot.");
//use a card for showing their data
var resultMessage = context.MakeMessage();
//resultMessage.AttachmentLayout = AttachmentLayoutTypes.Carousel;
resultMessage.Attachments = new List<Attachment>();
string ThankYouMessage;
```

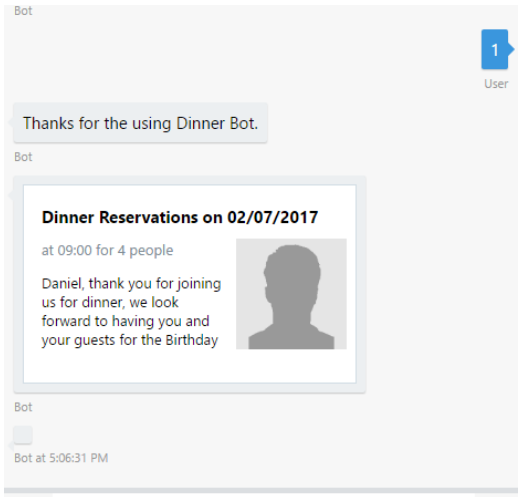
The next section just creates a custom thank you message depending on whether or not they are having a special occasion using the reservation variable from above.

```
if (reservation.SpecialOccasion == ReservationDialog.SpecialOccasionOptions.none)
{
    ThankYouMessage = reservation.Name +
        ", thank you for joining us for dinner, we look forward to having you and your guests.";
}
else
{
    ThankYouMessage = reservation.Name +
        ", thank you for joining us for dinner, we look forward to having you and your guests for the " +
        reservation.SpecialOccasion;
}
```

The final part (excluding the catches) creates a thumbnail card using the information from the form and posts it to the user.

```
ThumbnailCard thumbnailCard = new ThumbnailCard()
{
    Title = String.Format("Dinner Reservations on {0}", reservation.ReservationDate.ToString("MM/dd/yyyy")),
    Subtitle = String.Format("at {0} for {1} people", reservation.NumberOfDinners, reservation.ReservationTime.ToString("hh:mm")),
    Text = ThankYouMessage,
    Images = new List<CardImage>()
        {
            new CardImage() { Url = "https://upload.wikimedia.org/wikipedia/en/e/ee/Unknown-person.gif" }
        },
};

resultMessage.Attachments.Add(thumbnailCard.ToAttachment());
await context.PostAsync(resultMessage);
await context.PostAsync(String.Format(""));
```

Run your project and connect the emulator to test. If all works out fine, you should see the following when done.



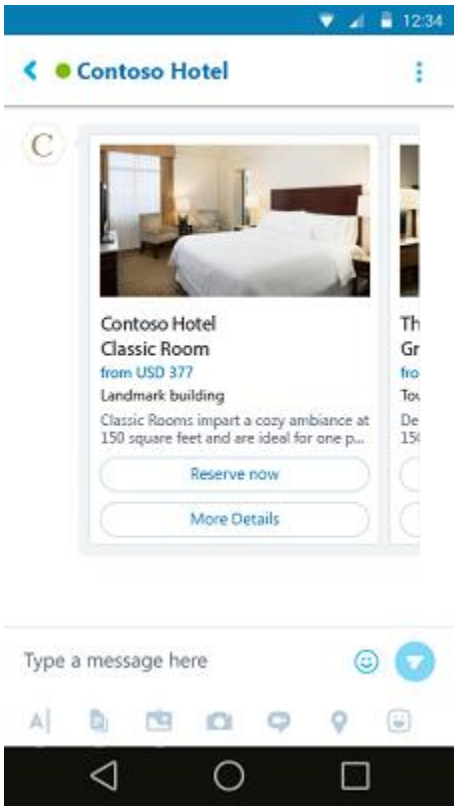
Detailed Steps
At the end of this exercise we utilized one hero card to show our reservation information. In the next exercise, we are going to explore this a bit more and look at Cards, Adaptive Cards and Carousels.

Exercise 4: Carousels, Cards, and Adaptive cards

In this exercise, we will look at some ways that we can present the data to the user so that it is a visually pleasant experience.

Detailed Steps

One of the most elegant ways to showcase data and options in a bot is by the use of a carousel. The sample below shows a Hero Card in a Carousel as depicted on Skype.



In the last exercise, we displayed a simple card all by itself. In this exercise, we will show you how to add cards to a carousel.

While it is not an exact fit to our project so far, we want to keep the code simple enough for you to use elsewhere so we will use the OpenTable public api to pull back a list of restaurants in a specific zip code. We will display these in a Carousel in 3 separate ways: A Hero Card, a Thumbnail Card, and an Adaptive Card.

The first thing we need to do is create a Dialog to use for searching. (For detailed instructions refer back to creating the HelloDialog above)

1. Right Click on your Dialogs Folder and Select **Add → Class**
2. Name the class **SearchRestaurantDialog**
3. Add the following using statements

```
using Microsoft.Bot.Builder.Dialogs ;
using Microsoft.Bot.Connector ;
```

4. Implement the **IDialog<object>** interface,
5. Make the class **[Serializable]**
6. Add the **async** qualifier to the **StartAsync** method

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using System.Web;
6  using Microsoft.Bot.Builder.Dialogs;
7  using Microsoft.Bot.Connector;
8
9  namespace DinnerBot.Dialogs
10 {
11     [Serializable]
12     public class SearchRestaurantDialog : IDialog<object>
13     {
14         public async Task StartAsync(IDialogContext context)
15         {
16             throw new NotImplementedException();
17         }
18     }
19 }
```

Next, let's put in some sample code to collect a zip code to use for our search. To do this we are going to use one of the **built-in prompt dialogs**. In this case, we will use number to insure they are giving us a number.

25. Add the following code to the StartAsync method (replacing the NotImplementedException code)

----- SNIP26----------

```
var responseMessage = "Please enter a zipcode";
            PromptDialog.Number(context, AfterChosenAsync,
responseMessage, "Sorry! that was not a number. Please enter a zip
code.",2);
```

This will ask the user to enter a zip code, if they don't enter a number it will ask again. If after two tries they still don't enter a number it will abort (default is 3). If they enter a number it will go to the callback called **AfterChosenAsync** so lets implement that.

26. Add the following code underneath the **StartAsync** method.

----- SNIP27----------

```
private async Task AfterChosenAsync(IDialogContext context,
IAwaitable<long> result)
```

```
{
    var message = await result;
    await context.PostAsync("You said " + message);
    context.Done<object>(null);
}
```

At this point, this method will capture the input (zip code) and echo it back to the user before exiting the dialog. We will use this for now to test before we add the call to OpenTable and implement the cards.

Now we need to wire up this dialog in the RootDialog

7. Open up your **RootDialog.cs** and add the following line to the top of the class near the other options.

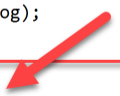
```
private const string ReservationOption = "Reserve Table";
private const string HelloOption = "Say Hello";
private const string SearchRestaurantsOption = "Search Restaurants";
```

8. Next we need to add an option to our prompt. In our PromptUser method add the following to the list of options.

```
private void PromptUser(IDialogContext context)
{
    PromptDialog.Choice(
    context,
    this.OnOptionSelected,
    // Present two (2) options to user
    new List<string>() { ReservationOption, HelloOption, SearchRestaurantsOption },
    String.Format("Hi {0}, are you looking for to reserve a table or Just say hello?",
}
```

9. In our case statement for **OnOptionSelected** and the following case.

```
//capture which option then selected
string optionSelected = await result;
switch (optionSelected)
{
    case ReservationOption:

        var form = new FormDialog<Reservation>(
        new Reservation(context.UserData.Get<String>("Name")),
        ReservationForm.BuildForm,
        FormOptions.PromptInStart,
        null);

        context.Call(form, this.ReservationFormComplete);
        break;

    case HelloOption:
        context.Call(new HelloDialog(), this.ResumeAfterUserHelloDialog);
        break;

    case SearchRestaurantsOption:
        context.Call(new SearchRestaurantDialog(), this.ResumeAfterUserHelloDialog);
        break;
```

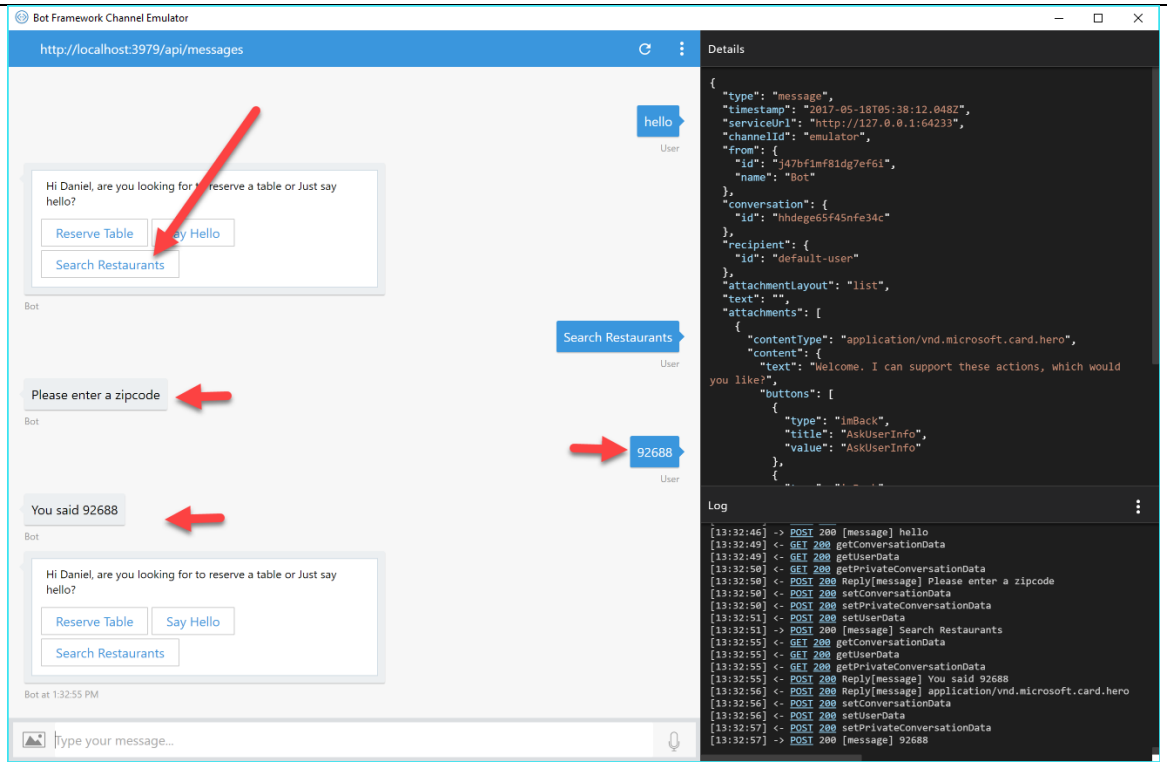


This will call our new dialog (SearchRestaurantDialog) if this option is selected.
Note that I have chosen to reuse the callback for the HelloDialog. The reason I am showing this is to illustrate the fact that both of these dialogs return nothing so there is no need for a specific callback here. Obviously, if you were doing this in a real project you would name the callback something more generic like DefaultResumeAfterDialog or something similar.

Now lets test what we have so far to make sure it is working.

10. Run the project.
11. Refresh the emulator
12. Type hello and then enter a zip code when asked.

You should see the following if all worked out correctly.



Now we need to wire up our api call to OpenTable and show the data in cards. Since the point of this exercise is to learn cards we are going to do this all in the callback method we just created.

Lets start in the AfterChosenAsync method.

13. Delete the line where we just echo back the zip code since we will use the zip code for the API call

```
private async Task AfterChosenAsync(IDialogContext context, IAwaitable<long> result)
{
    var message = await result;
    await context.PostAsync("You said " + message);
    context.Done<object>(null);
}
```

14. Add the following Using Statement

using System.Net.Http;
using DinnerBot.Models;

15. Paste the following code where the await context = PostAsync(“You said “ + message used to be)

----- SNIP28----------

```
//Create Message
var reply = context.MakeMessage();
//Set reply type to Carousel
reply.AttachmentLayout = AttachmentLayoutTypes.Carousel;

//Make the call to the OpenTable API
using (var client = new HttpClient())
{
    try
    {
       string url =
"https://opentable.herokuapp.com/api/restaurants?zip=" + message;
       HttpResponseMessage response = await client.GetAsync(url);
        if (response.IsSuccessStatusCode)
        {
            //retrieve response
            var json = await response.Content.ReadAsStringAsync();
            //create a object from the json
            var des =
(RootObject)Newtonsoft.Json.JsonConvert.DeserializeObject(json,
typeof(RootObject));

            //Create a list of cards to use for the data coming back.
These are of type Attachment
            List<Attachment> cards = new List<Attachment>();

            //Loop through the results and turn them into cards
            //Note:  I limit them to 10 because Skype has a limit of 10.
If you send more none will show.
            foreach (var info in des.restaurants.Take(10))
            {
                //This dataset has images with it but they dont come
back to the emultor so we are using
                //a default opentable image
                var image =
"http://media.opentable.com/about/images/logos/ogimage.jpg";
//info.image_url;
                //Call our card util to return the type of card we want.
                Attachment card = Utils.Cards.GetHeroCard(
                    info.name,
                    info.address,
                    info.city,
                    new CardImage(url: image),
                    new CardAction(ActionTypes.OpenUrl, "Learn more",
value: info.reserve_url)
                    );
                cards.Add(card);
            }
            //when done add the cards to the reply
```

```
            reply.Attachments = cards;
            //post the reply (The cards in a carousel)
            await context.PostAsync(reply);
            //exit dialog
            context.Done<object>(null);
        }
    }
    catch (Exception ex)
    {

        string myerror = ex.ToString();
    }

}
```

In this code we are:
Creating a Message
Setting its layout to Carousel
Calling the API and retrieving the json result
turning it into a .net object and looping through to create the cards

It is commented, please look thorough to see all that it is doing.

Next there are a few classes we need to complete this. We need the models of the objects that are returned from open table (RootObject and Restaurant) and we need a card util. Lets create the models first. These were just create by using http://json2csharp.com/ so we will not discuss them we will just create them.

16. Right click on the Models folder and **Add → Class** and name it **RootObject.cs**
17. Replace everything in that file with the following code.

----- SNIP29----------

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

namespace DinnerBot.Models
{
    public class RootObject
    {
        public int total_entries { get; set; }
        public int per_page { get; set; }
        public int current_page { get; set; }
        public List<Restaurant> restaurants { get; set; }
```

```
    }
}
```

18. Next, right click on the Models folder select **Add → Class** and name it **Restaurant.cs**
19. Replace everything in the file with the following code.

----- SNIP30----------

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

namespace DinnerBot.Models
{
    public class Restaurant
    {
        public int id { get; set; }
        public string name { get; set; }
        public string address { get; set; }
        public string city { get; set; }
        public string state { get; set; }
        public string area { get; set; }
        public string postal_code { get; set; }
        public string country { get; set; }
        public string phone { get; set; }
        public double lat { get; set; }
        public double lng { get; set; }
        public int price { get; set; }
        public string reserve_url { get; set; }
        public string mobile_reserve_url { get; set; }
        public string image_url { get; set; }
    }
}
```

Finally, we need to create the utility for creating the cards. We have abstracted this out into its own class in order to see it all by itself and not cluttered up with other code. We need to create a folder for our cards utility class.

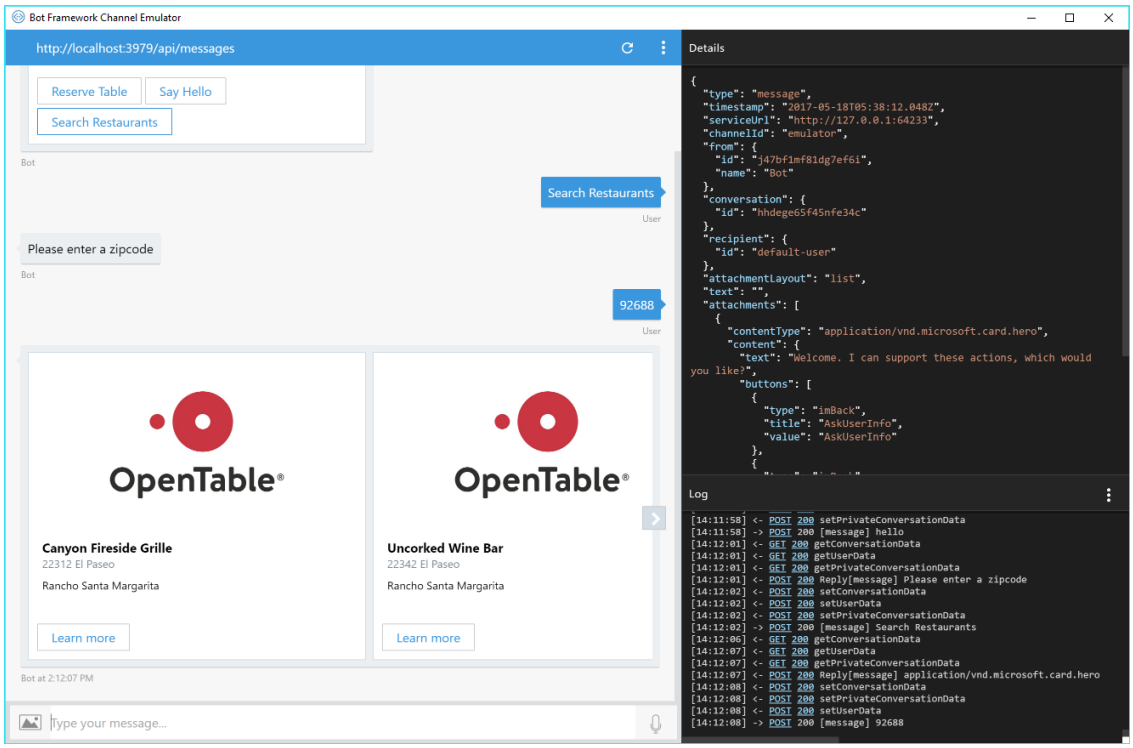
20. Right Click on the project and select **Add → New Folder** and name it **Utils**
21. Right Click on the **Utils** Folder and select Add → Class and name it **Cards**
22. Replace everything in the file with the following code.

----- SNIP31----------

```
using Microsoft.Bot.Connector;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace DinnerBot.Utils
{
    public class Cards
    {
        //Create HeroCard method that takes in the data needed to
construct the card, title, subtitle, image, etc..
        public static Attachment GetHeroCard(string title, string
subtitle, string text, CardImage cardImage, CardAction cardAction)
        {
            //Create a new herocard
            var heroCard = new HeroCard
            {
                //set the properties of the card
                Title = title,
                Subtitle = subtitle,
                Text = text,
                Images = new List<CardImage>() { cardImage },
                Buttons = new List<CardAction>() { cardAction },
            };

            //return it as an attachment
            return heroCard.ToAttachment();
        }

        public static Attachment GetThumbnailCard(string title, string
subtitle, string text, CardImage cardImage, CardAction cardAction)
        {
            var thumbNailCard = new ThumbnailCard
            {
                Title = title,
                Subtitle = subtitle,
                Text = text,
                Images = new List<CardImage>() { cardImage },
                Buttons = new List<CardAction>() { cardAction },
            };

            return thumbNailCard.ToAttachment();
        }

    }
}
```

In this code we are creating an instance of a hero card and setting its properties before sending it back as an attachment to the caller. There is one method for the Hero Card and one for the Thumbnail card. The other card types could be created in the same way.

That is all that we need to do. Let's run our project to test it out. If all was done correctly you should see this. (Making sure you use a zip code that works)



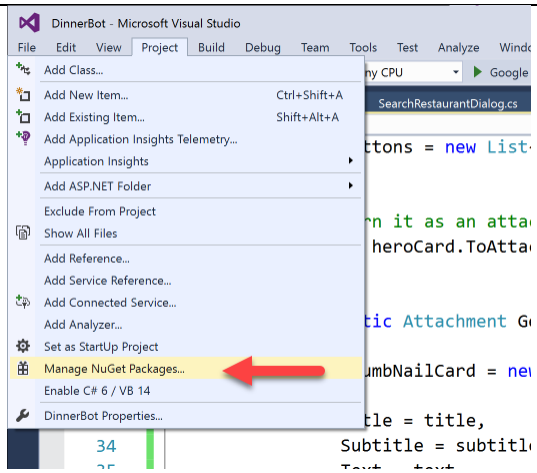
We set the project to use the **GetHeroCard** method. If you would like to see what the thumbnails look like you can change this line in **SearchRestaurantDialog**.cs to call **GetThumbnailCard**

```
//a default opentable image
var image = "http://media.opentable.com/about/images/logos/ogimage.jpg"; //info.image_url;
//Call our card util to return the type of card we want.
Attachment card = Utils.Cards.GetHeroCard(
    info.name,
    info.address,
```

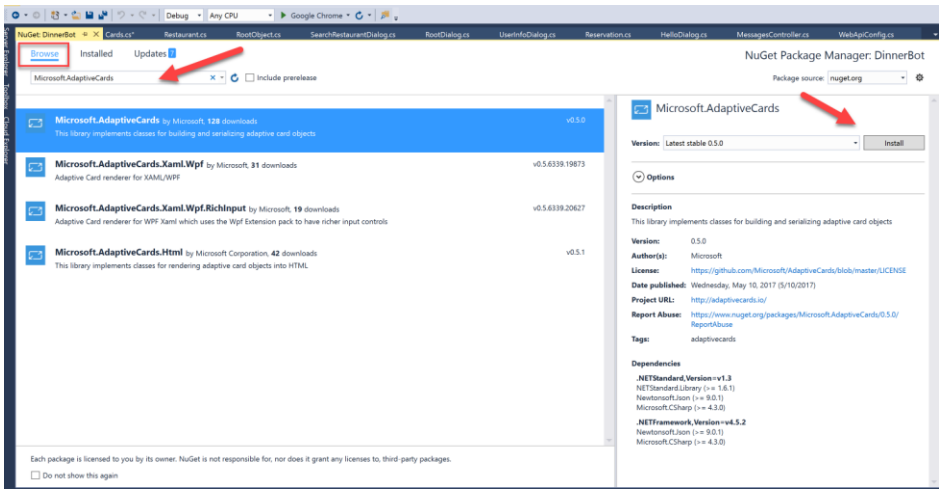
Now all of the static cards are great, but if you want more flexibility you will want to utilize Adaptive Cards. Adaptive cards lets you create cards in the format and layout that you would like and makes sure they are formatted correctly across channes.

In order to user adaptive cards you will need to add the nuget package for adaptive cards.

23. Go to **Project → Manage NuGet Packages**



24. Click on the **Browse** tab and search for **Microsoft.AdaptiveCards**, then click **Install** to install the package. (you can of course use the Package Manager Console to install as well - PM> Install-Package Microsoft.AdaptiveCards)



25. Once installed, go back to the Cards.cs file and add the following using statement

using AdaptiveCards;

26. Finally, add the following method GetAdaptiveCard to the Cards.cs file below the GetThumbnailCard Method.

This is arguably a bunch of code but I wanted to show you two different ways to code your ColumnSets, Columns, TextBlocks etc..

----- SNIP32----------

```
public static Attachment GetAdaptiveCard(string title, string subtitle,
string text, CardImage cardImage, CardAction cardAction)
```

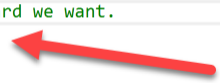
```
{
    var adaptiveCard = new AdaptiveCard
    {
        BackgroundImage = "https://thumbs.dreamstime.com/z/perspective-
wood-over-blurred-restaurant-bokeh-background-foods-drinks-product-
display-montage-55441300.jpg",
        Body = new List<CardElement>
        {
            new ColumnSet()
            {
                Columns = new List<Column>()
                {
                    new AdaptiveCards.Column()
                    {
                        Size = "3",
                        Items = new List<AdaptiveCards.CardElement>()
                        {
                            new TextBlock() { Text = title, Size =
TextSize.Large, Weight = TextWeight.Bolder },
                            new TextBlock() { Text = subtitle},
                            new FactSet()
                            {
                                Facts = new List<AdaptiveCards.Fact>()
                                {
                                    new AdaptiveCards.Fact() {Title =
"Fact 1", Value = "Value 1" },
                                    new AdaptiveCards.Fact() {Title =
"Fact 2", Value = "Value 2" }
                                }
                            },
                            new ChoiceSet()
                            {
                                Id = "Times",
                                Style = ChoiceInputStyle.Compact,
                                Choices = new List<Choice>()
                                {
                                    new Choice() { Title = "6 PM", Value
= "6", IsSelected = true },
                                    new Choice() { Title = "7 PM", Value
= "7" },
                                    new Choice() { Title = "8 PM", Value
= "8" }
                                }

                            }
                        }
                    },
                    new AdaptiveCards.Column()
                    {
```

```
                        Items = new List<AdaptiveCards.CardElement>()
                        {
                           new Image(){Url = cardImage.Url,Size =
ImageSize.Stretch}
                        }
                    }

                }
            }

        }
    };

    /*////////////////////////////////////////////////////////////////
    Alternate way to create your cards, columns, textblocks, etc..
    ////////////////////////////////////////////////////////////////*/

    // ColumnSet set = new ColumnSet();
    // Column c1 = new Column()
    // {

    // };
    // Column c2 = new Column();
    // set.Columns.Add(c1);
    // set.Columns.Add(c2);

    // c1.Items.Add(new TextBlock()
    // {
    //     Text = title,
    //     Size = TextSize.Large,
    //     Weight = TextWeight.Bolder
    // });

    // c1.Items.Add(new TextBlock()
    // {
    //     Text = subtitle
    // });
    // c1.Items.Add(new FactSet()
    // {
    //     Facts = new List<AdaptiveCards.Fact>()
    //     {
    //         new AdaptiveCards.Fact() {Title = "Fact 1", Value =
"Value 1" },
    //         new AdaptiveCards.Fact() {Title = "Fact 2", Value =
"Value 2" }
    //     }
    // });

    // // Add list of choices to the card.
```

```
    //c1.Items.Add(new ChoiceSet()
    // {
    //     Id = "snooze",
    //     Style = ChoiceInputStyle.Compact,
    //     Choices = new List<Choice>()
    //     {
    //         new Choice() { Title = "5 minutes", Value = "5",
IsSelected = true },
    //         new Choice() { Title = "15 minutes", Value = "15" },
    //         new Choice() { Title = "30 minutes", Value = "30" }
    //     }
    // });
    // c2.Items.Add(new Image()
    // {
    //     Url = cardImage.Url,
    //     Size = ImageSize.Stretch
    // });

    // card.Body.Add(set);


    // Add text to the card.
    //card.Body.Add(new TextBlock()
    //{
    //    Text = title,
    //    Size = TextSize.Large,
    //    Weight = TextWeight.Bolder
    //});

    // Add text to the card.
    //card.Body.Add(new TextBlock()
    //{
    //    Text = subtitle
    //});
    //card.Body.Add(new Image()
    //{
    //    Url = cardImage.Url,
    //    Size = ImageSize.Medium
    //});


    // Create the attachment.
    Attachment attachment = new Attachment()
    {
        ContentType = AdaptiveCard.ContentType,
        Content = adaptiveCard
    };
    return attachment;
}
```

Once again, take the time to look through the code to see what it does. The final product could be a bit nicer but I wanted to show you how to stack columns so that it could be understood without too much bloated code. To see this in action, you will need to change the same line in **SearchRestaurantDialog** to **GetAdaptiveCard**

```
//a default opentable image
var image = "http://media.opentable.com/about/images/logos/ogimage.jpg"; //info.image_url;
//Call our card util to return the type of card we want.
Attachment card = Utils.Cards.GetHeroCard(
    info.name,
    info.address,
```



One last note, the method for GetAdaptiveCard follows the same signature as the other methods for ease of demo but since you can add whatever you want to and adaptive care you would need to change what properties are coming in to feed it.

Run your project to see the adaptive cards.

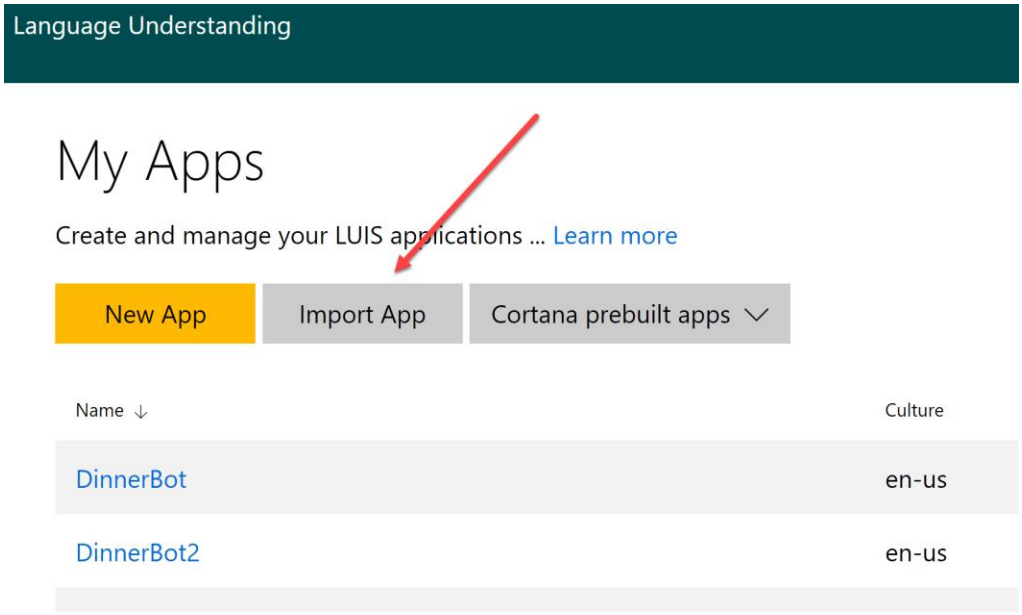
That's it for this Exercise. Next, we will look into Natural Language Process with LUIS to add some AI to your project.

Exercise 5: Using Intent Dialogs (LUIS)

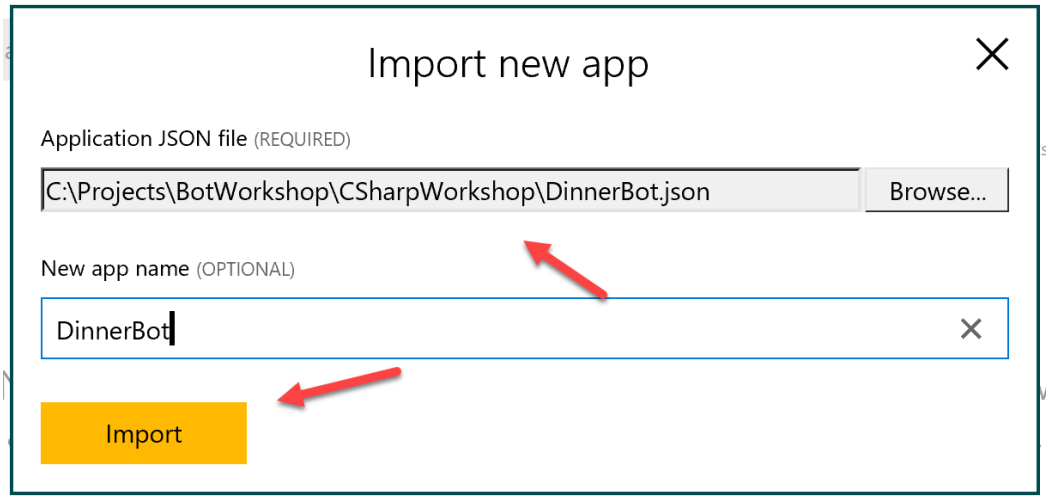
In this exercise, we will import a LUIS Model that will handle questions coming from the users and route them to the appropriate Dialogs. We will not be creating the model but importing an already existing model. If you would like to learn how to create your own model you can find great tutorials and walkthroughs here : https://www.luis.ai/Help

Detailed Steps

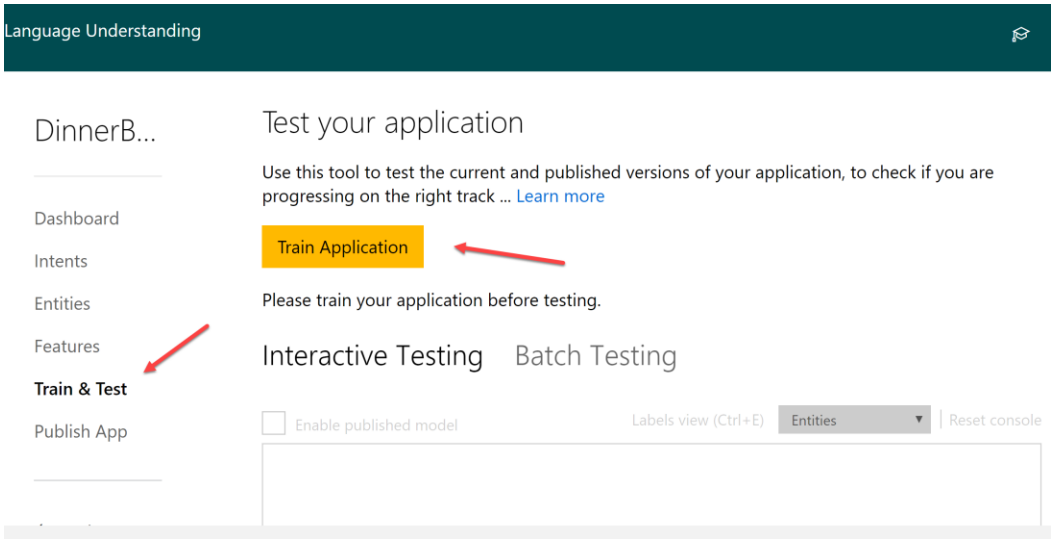
1. Sign on to http://www.LUIS.ai. You should have set this up in the first exercise, if not go back to the first section.
2. From your dashboard Select → **Import App**



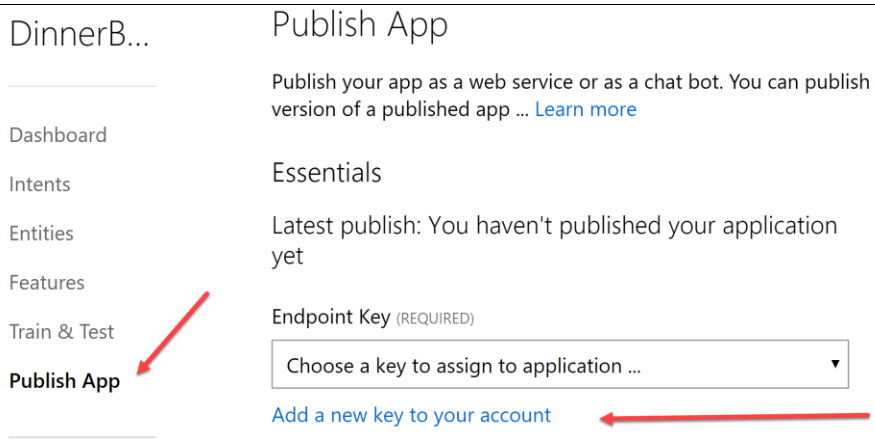
3. Click Choose File to import the existing LUIS app. The file will be called **DinnerBot.json** and you will find it in the **BotWorkshop\CSharpWorkshop** folder of the git repository you cloned. Name it **DinnerBot** and click on import.



4. The next thing we need to do is train the model. Click on Train & Test on the left hand menu and then click on the Train Application button.

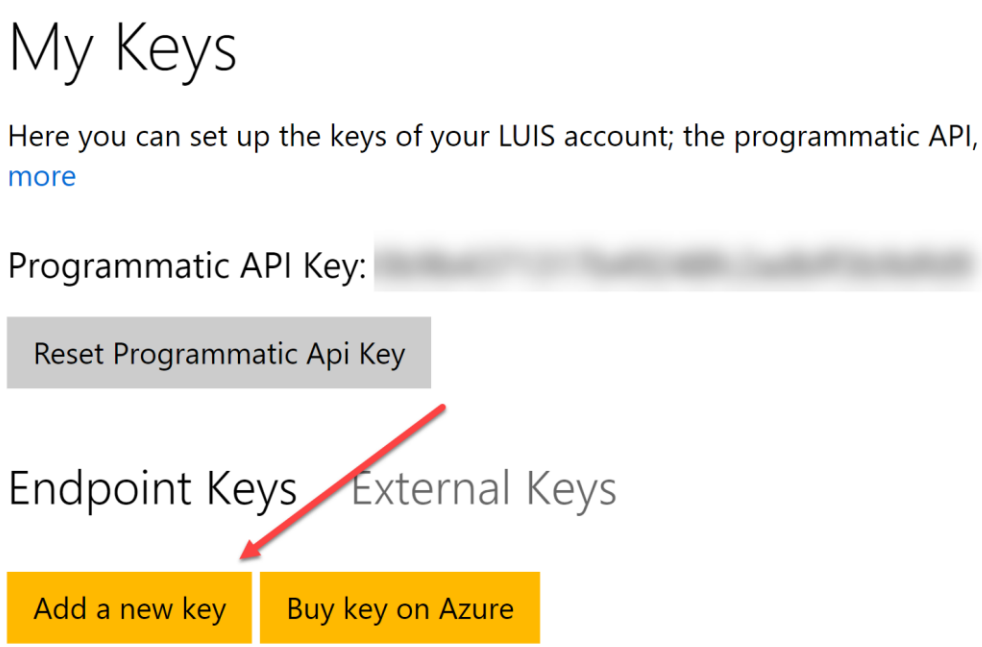


5. Once it is trained, we need to publish the model. On the left of the screen click on the Publish App link.

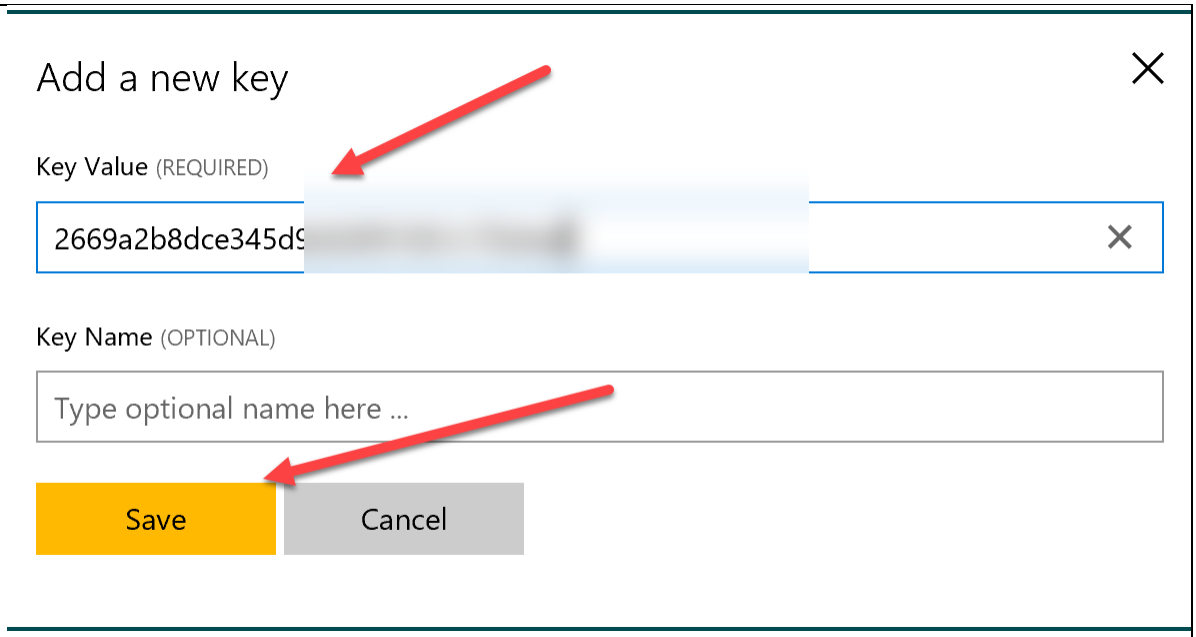


The first thing you need to do is add a key (if this not your first time using LUIS, your key will be found in the dropdown)

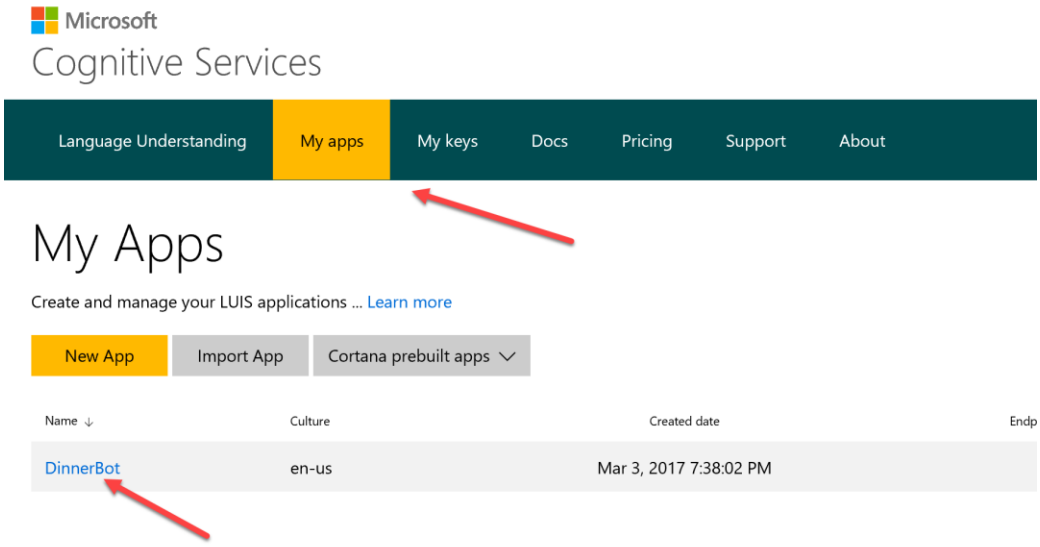
6. Highlight and copy they key next to "Programmatic API Key:" Once you have that copied to your clipboard, Click on the Add a new key button



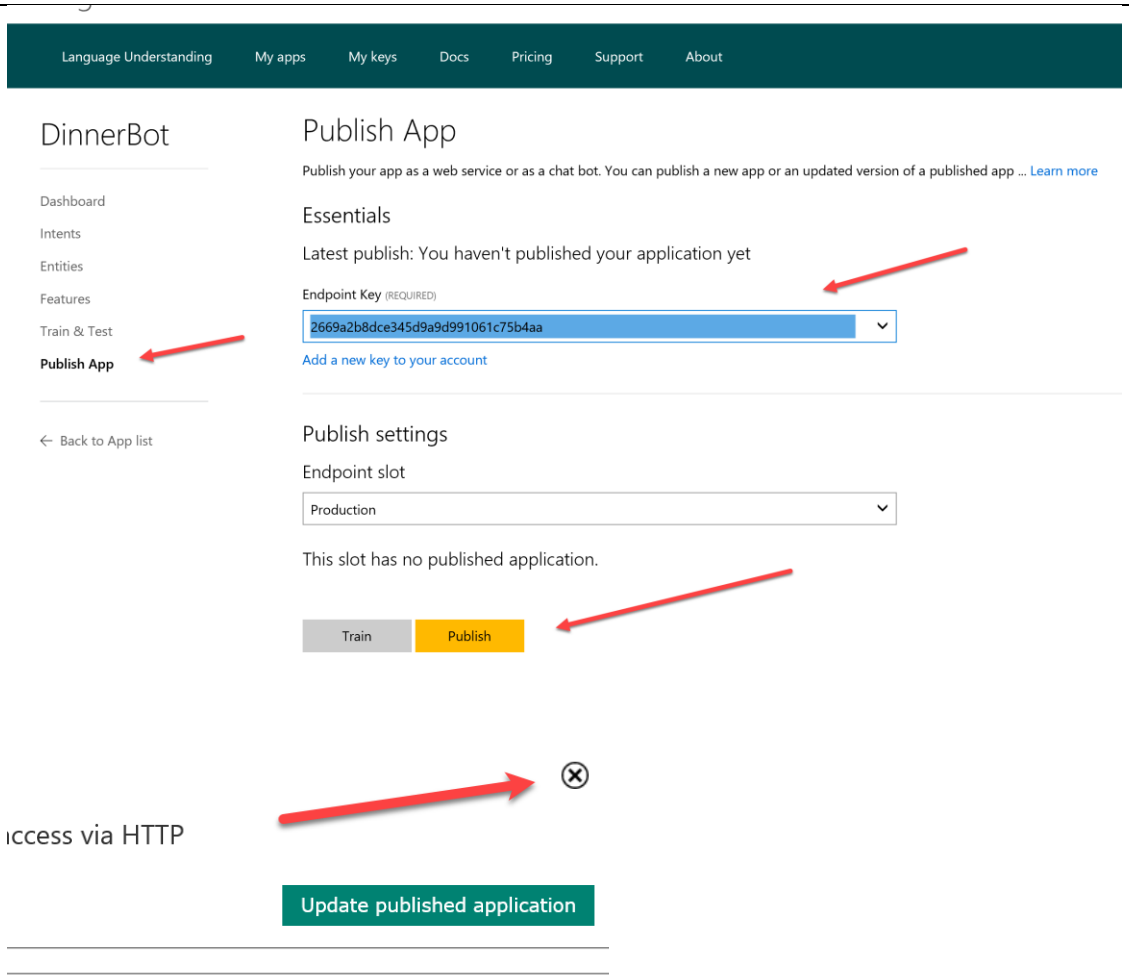
7. Paste the key into the Key Value box and click on save (you can optionally name it if you would like)



Click on MY Apps on the Top Bar and then click on the DinnerBot link to bring it up so we can publish it.



Next, click on Publish App in the left menu, select your key from the Endpoint Key dropdown, and click on the Publish button.



Leave the LUIS.ai website open, we will need some data from it in a moment.

Now we need to modify our RootDialog in order to have it work with LUIS.

8. Open the RootDialog.cs file and add the following Using statements to the top of the file.

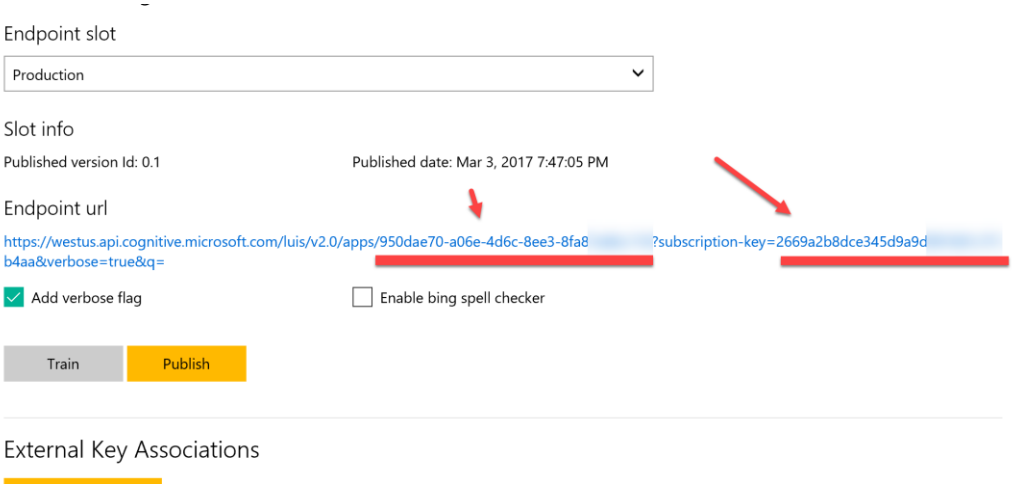
```
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
```

9. Next, add the [LuisModel] attribute to the top of the class below the [Serializable] attribute

```
namespace DinnerBot.Dialogs
{
    [Serializable]
    [LuisModel("modelID", "subscriptionKey")]
    public class RootDialog : IDialog<object>
    {
```

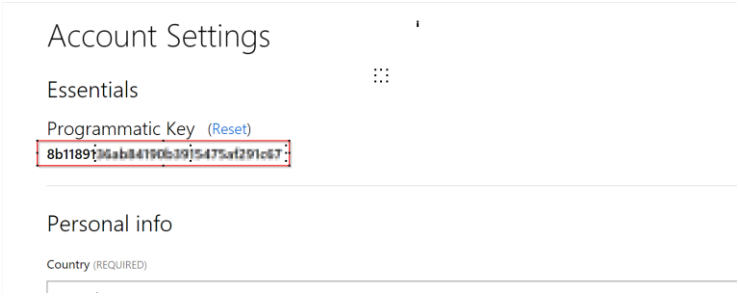
This will allow us to integrate with LUIS. We just need to add the **modelID** and Subscription key. We can get these from the LUIS.ai website.

10. Go back to the **LUIS.ai** website (Sign on if you need to) and open up your **DinnerBot** application. You will find the **model ID** right under where you clicked on publish.



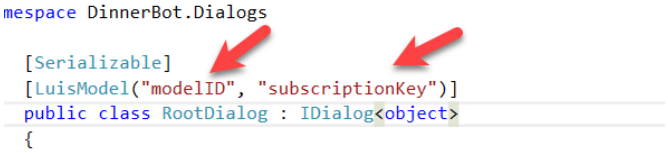
In the Endpoint url, you will find the Model ID right after /apps/ in the url.

11. Still in the **LUIS.ai** website, click on **your name** in the upper right hand corner, this will take you to the **Account Settings** page.



In the **Account Settings** page, you will see the **Programmatic Key**, which you should use as your subscription key.

12. Back in the **RootDialog.cs** file. Replace the strings modelID and **subscriptionKey** with the values you just retrieved. (Remember modelID is the same as App ID from LUIS website, and subscriptionKey is your Programmatic Key from the LUIS website.)



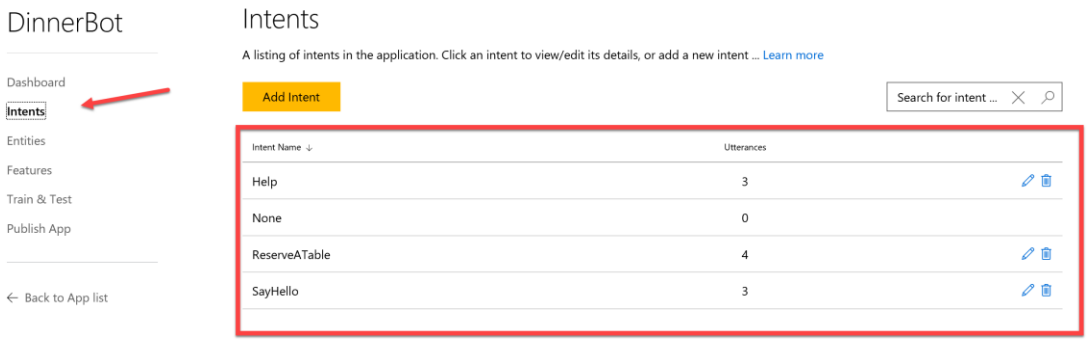
```
[Serializable]
[LuisModel("d15aae24-[redacted]", "0b9b[redacted]")]
public class RootDialog : IDialog<object>
```

We also need to change the interface that our **RootDialog** inherits from. Change it from **IDialog**<> to **LuisDialog**<>

```
public class RootDialog : IDialog<object>
{
```

```
[LuisModel("[redacted]")]
public class RootDialog : LuisDialog<object>
{
```

Now we are ready to add our intents. This will fundamentally change how our RootDialog works. What we need when working with LUIS is methods that map (using attributes) to the intents form LUIS. So if we look at our Intents in LUIS, we need to map to the following Intents



In the **RootDialog.cs** file, remove the **StartAsync** method and replace it with the following code. One again, it's a lot of code but we will step through it.

This code **REPLACES** the **StartAsync** method in RootDialog. We don't need it since we are not implementing IDialog<>

------SNIP26----------

```
[LuisIntent("")]
[LuisIntent("None")]
public async Task None(IDialogContext context, LuisResult result)
{
    string message = $"Sorry, I did not understand '{result.Query}'";
    await context.PostAsync(message);
    context.Wait(MessageReceived);
}
```

```
[LuisIntent("ReserveATable")]
public async Task ReserveATable(IDialogContext context, LuisResult result)
{
    try
    {
        await context.PostAsync("Great, lets book a table for you. You will need to
provide a few details.");
var form = new FormDialog<Reservation>(
new Reservation(context.UserData.Get<String>("Name")),
ReservationForm.BuildForm,
FormOptions.PromptInStart,
null);

context.Call(form, this.ReservationFormComplete);
    }
    catch (Exception)
    {
        await context.PostAsync("Something really bad happened. You can try again
later meanwhile I'll check what went wrong.");
        context.Wait(MessageReceived);
    }
}

[LuisIntent("SayHello")]
public async Task SayHello(IDialogContext context, LuisResult result)
{
    context.Call(new HelloDialog(), this.ResumeAfterOptionDialog);
}
[LuisIntent("Help")]
public async Task Help(IDialogContext context, LuisResult result)
{
    await context.PostAsync("Insert Help Dialog here");
    context.Wait(MessageReceived);
}
```

The first method has attributes that match a not found Luis Intent and one that is captured by None. Note that the result of this method is not a **LuisResult**. Also notice the **context.Wait**, the callback is **MessageReceived**. This is not something we write, but is part of the **LuisDialog**. It sets it ready for another Luis request.

```
[LuisIntent("")]
[LuisIntent("None")]
public async Task None(IDialogContext context, LuisResult result)
{
    string message = $"Sorry, I did not understand '{result.Query}'";
    await context.PostAsync(message);
    context.Wait(MessageReceived);
}
```

Next is the main one the ReserveATable intent. The code inside here is exactly the same as we used in the last exercise except that it is arrived by someone asking LUIS instead of answering a prompt.

```
[LuisIntent("ReserveATable")]
public async Task ReserveATable(IDialogContext context, LuisResult result)
{
    try
    {
        await context.PostAsync("Great, lets book a table for you. You will need to provide a few details.");
        var reservationForm = new FormDialog<ReservationDialog>(new ReservationDialog(), ReservationDialog.BuildForm, FormOptio
        context.Call(reservationForm, ReservationFormComplete);
    }
    catch (Exception)
    {
        await context.PostAsync("Something really bad happened. You can try again later meanwhile I'll check what went wrong.")
        context.Wait(MessageReceived);
    }
}
```

The last two implement the hello and help (which we did not implement)

```
[LuisIntent("SayHello")]
public async Task SayHello(IDialogContext context, LuisResult result)
{
    context.Call(new HelloDialog(), this.ResumeAfterOptionDialog);
}
[LuisIntent("Help")]
public async Task Help(IDialogContext context, LuisResult result)
{
    await context.PostAsync("Insert Help Dialog here");
    context.Wait(MessageReceived);
}
```

That's it, run your project and fire up the emulator. You can now try to ask for a reservation in different ways to see how LUIS handles it. Try things like "book a table" or "I need a table" if they don't work, go back up to LUIS and train it some more to recognize additional statements.

Additional Resources

Copyright