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

Hands-on Lab Manual

Table of Contents

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#

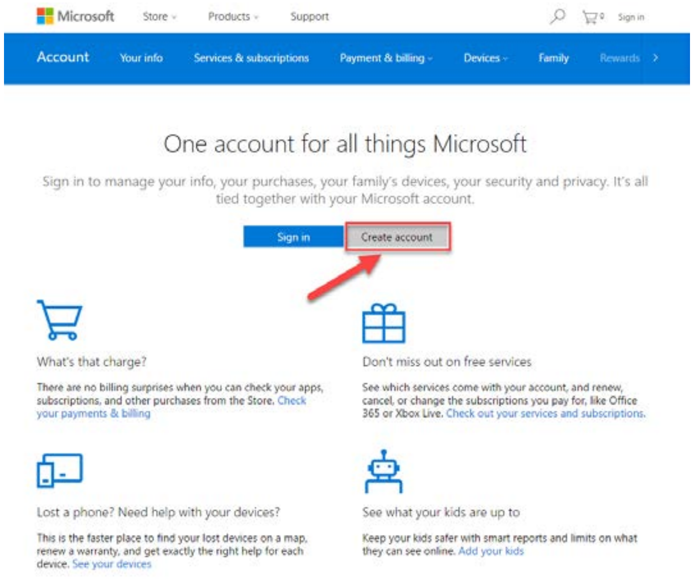
Lab Scenarios

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.botframework.com/en-us/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 Create account**.



One account for all things Microsoft

Sign in to manage your info, your purchases, your family's devices, your security and privacy. It's all tied together with your Microsoft account.

What's that charge?

There are no billing surprises when you can check your apps, subscriptions, and other purchases from the Store. Check your payments & billing

Don't miss out on free services

See which services come with your account, and renew, cancel, or change the subscriptions you pay for, like Office 365 or Xbox Live. Check out your services and subscriptions.

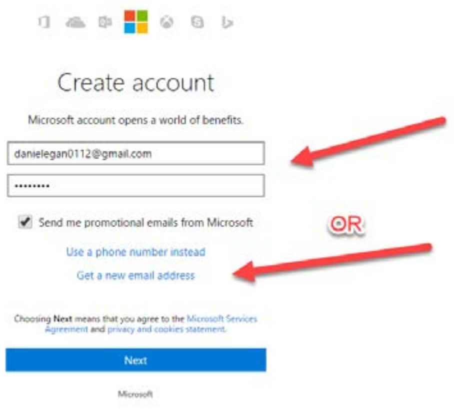
Lost a phone? Need help with your devices?

This is the faster place to find your lost devices on a map, renew a warranty, and get exactly the right help for each device. See your devices

See what your kids are up to

Keep your kids safer with smart reports and limits on what they can see online. Add your kids

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


Create account

Microsoft account opens a world of benefits.

danielegan0112@gmail.com

••••••••

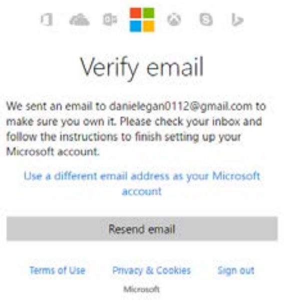
[✓] Send me promotional emails from Microsoft

Use a phone number instead

Get a new email address

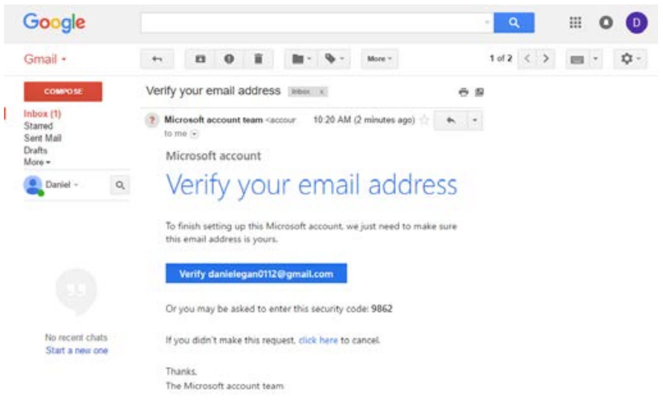
Choosing **Next** means that you agree to the Microsoft Services Agreement and privacy and cookies statement.

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


Verify email

We sent an email to danielegan0112@gmail.com to make sure you own it. Please check your inbox and follow the instructions to finish setting up your Microsoft account.

Use a different email address as your Microsoft account



Verify your email address Inbox x

Microsoft account team <accour 10:20 AM (2 minutes ago)
to me

Microsoft account

Verify your email address

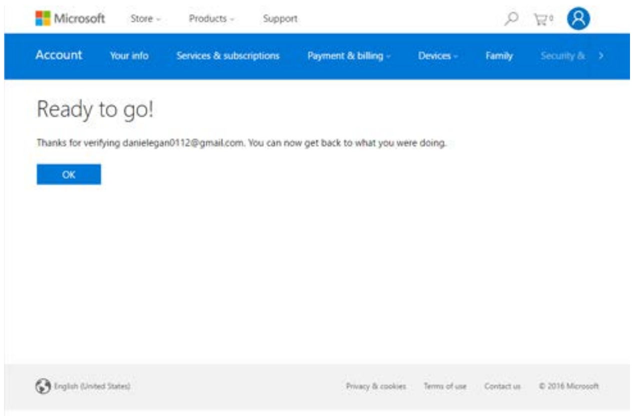
To finish setting up this Microsoft account, we just need to make sure this email address is yours.

Verify danielegan0112@gmail.com

Or you may be asked to enter this security code: **9862**

If you didn't make this request, click here to cancel.

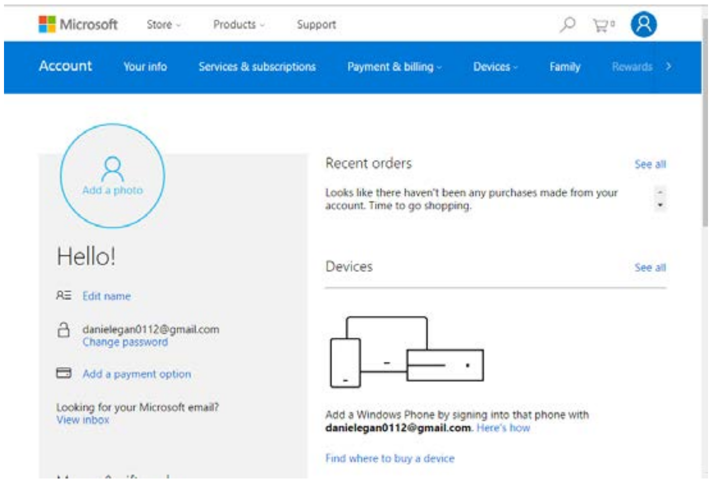
Thanks,
The Microsoft account team



Ready to go!

Thanks for verifying danielegan0112@gmail.com. You can now get back to what you were doing.

- Either path will take you to this screen

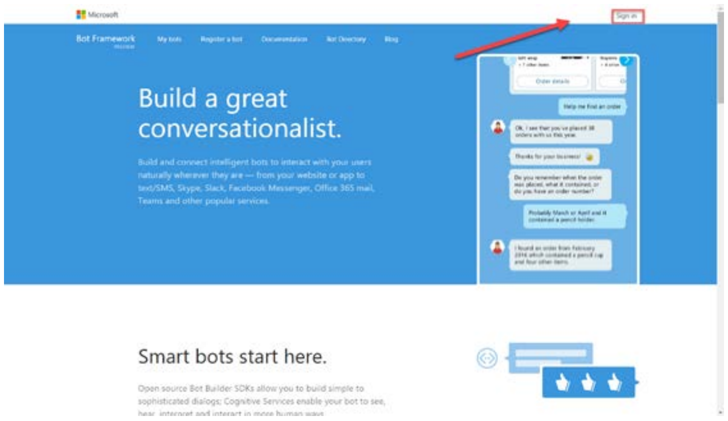


Looks like there haven't been any purchases made from your account. Time to go shopping.

Looking for your Microsoft email? View inbox

Add a Windows Phone by signing into that phone with **danielegan0112@gmail.com**. Here's how

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



Build a great conversationalist.

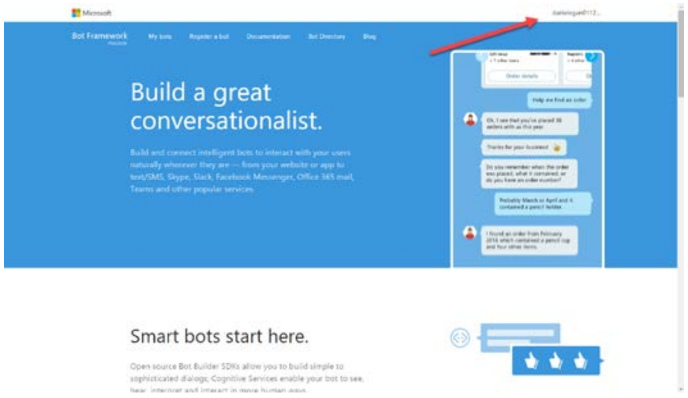
Build and connect intelligent bots to interact with your users naturally wherever they are — from your website or app to text/SMS, Skype, Slack, Facebook Messenger, Office 365 mail, Teams and other popular services.

Smart bots start here.

Open source Bot Builder SDKs allow you to build simple to sophisticated dialogs; Cognitive Services enable your bot to see, hear, interpret and interact in more human ways

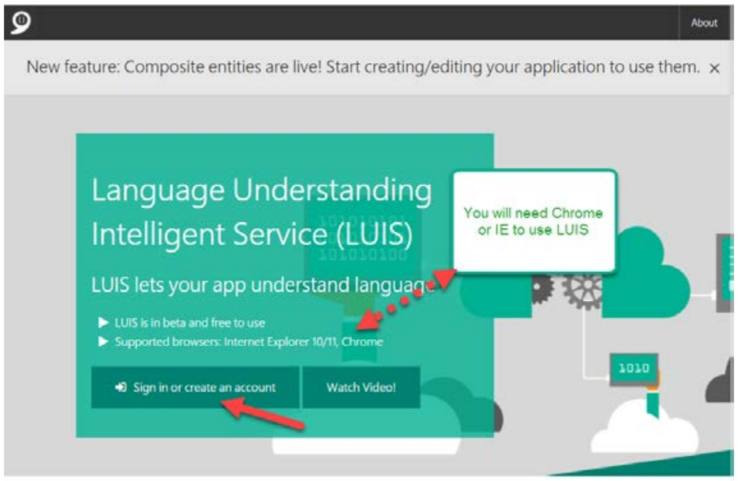
   - If you are using the same browser that you used to create your Microsoft ID then you will be signed in automatically, otherwise you will need to use the

ID you just created to sign in.

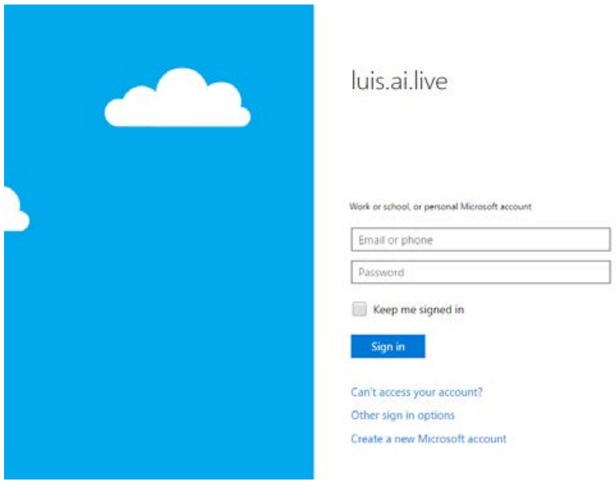


- You can leave this window open, we will be using it later.

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


About
New feature: Composite entities are live! Start creating/editing your application to use them.
Language Understanding Intelligent Service (LUIS)
LUIS lets your app understand language
LUIS is in beta and free to use
Supported browsers: Internet Explorer 10/11, Chrome
Sign in or create an account
Watch Video!
You will need Chrome or IE to use LUIS

   - Sign in with your Microsoft account


luis.ai.live

Work or school, or personal Microsoft account

Email or phone

Password

Keep me signed in

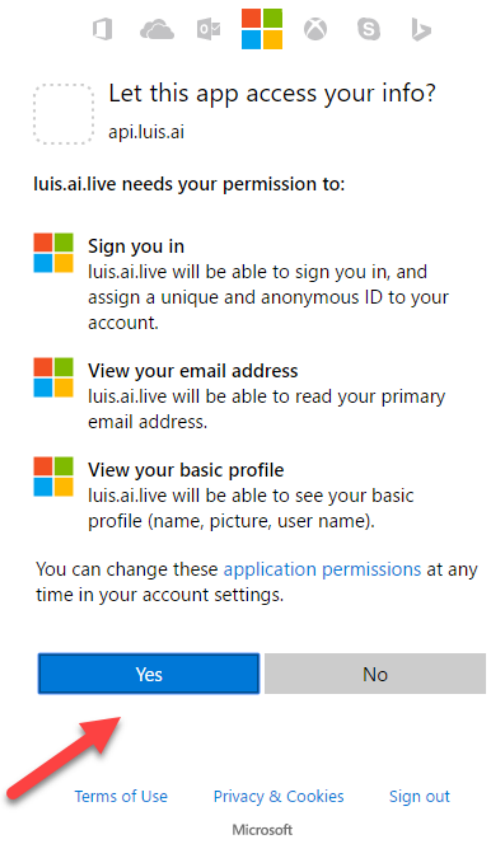
Sign in

Can't access your account?

Other sign in options

Create a new 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.


Let this app access your info?

api.luis.ai

luis.ai.live needs your permission to:

Sign you in
luis.ai.live will be able to sign you in, and assign a unique and anonymous ID to your account.

View your email address
luis.ai.live will be able to read your primary email address.

View your basic profile
luis.ai.live will be able to see your basic profile (name, picture, user name).

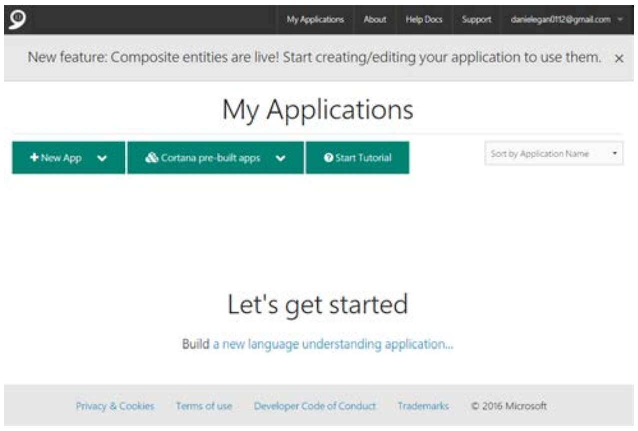
You can change these application permissions at any time in your account settings.

Yes No

Terms of Use Privacy & Cookies Sign out

Microsoft

- You can walk through the quick walkthrough if you would like or click the x to close it. When finished, your screen should look like below.



New feature: Composite entities are live! Start creating/editing your application to use them.

My Applications

Let's get started

Build a new language understanding application...

- 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.

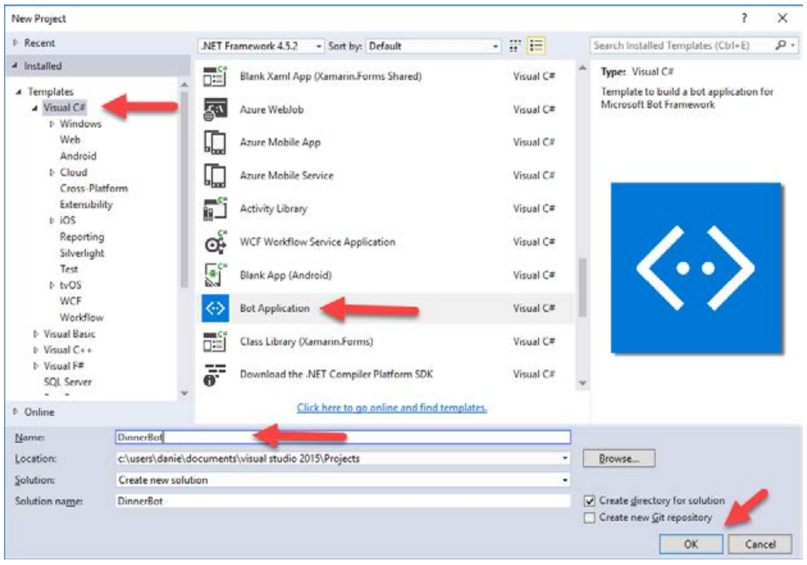
Exercise 1: Basic Bot using BotBuilder

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 don't 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 prerequisites above)

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

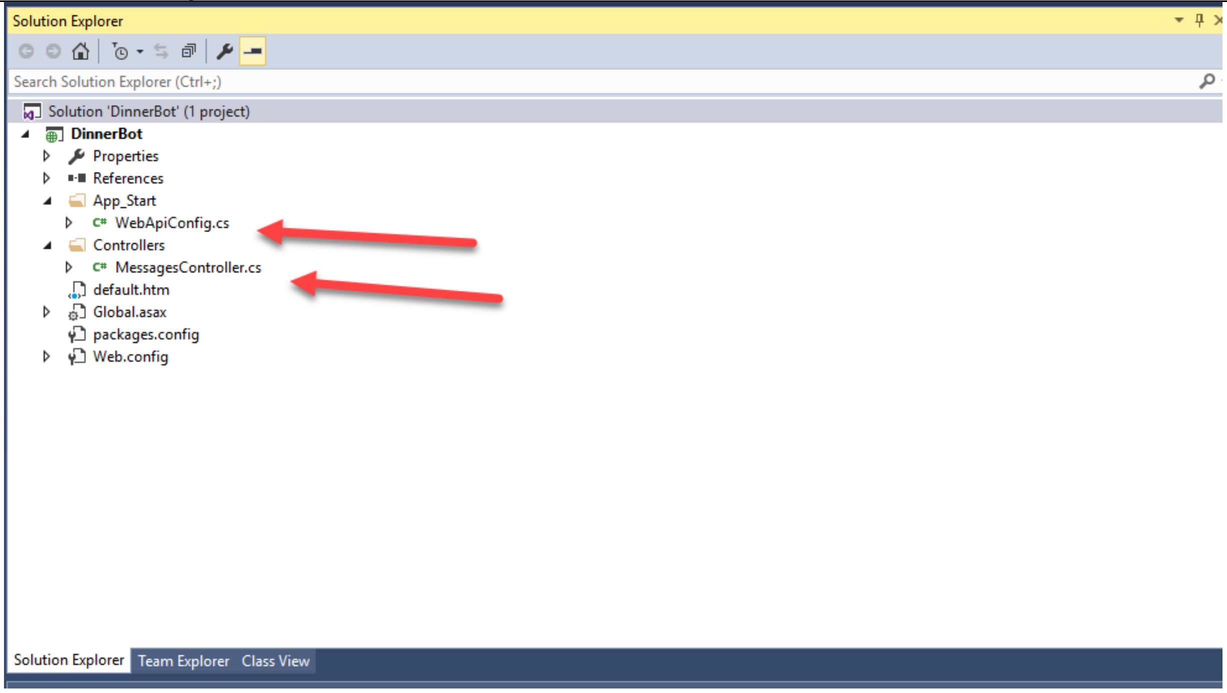

New Project
?
Recent
Installed
Templates
Visual C#
Windows
Web
Android
Cloud
Cross-Platform
Extensibility
iOS
Reporting
Silverlight
Test
tvOS
WCF
Workflow
Visual Basic
Visual C++
Visual F#
SQL Server
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank Xaml App (Xamarin.Forms Shared) Visual C#
Azure WebJob Visual C#
Azure Mobile App Visual C#
Azure Mobile Service Visual C#
Activity Library Visual C#
WCF Workflow Service Application Visual C#
Blank App (Android) Visual C#
Bot Application Visual C#
Class Library (Xamarin.Forms) Visual C#
Download the .NET Compiler Platform SDK Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#
Template to build a bot application for Microsoft Bot Framework

Name: DinnerBot
Location: c:\users\danie\documents\visual studio 2015\Projects
Solution: Create new solution
Solution name: DinnerBot
[✓] Create directory for solution
[ ] Create new Git repository

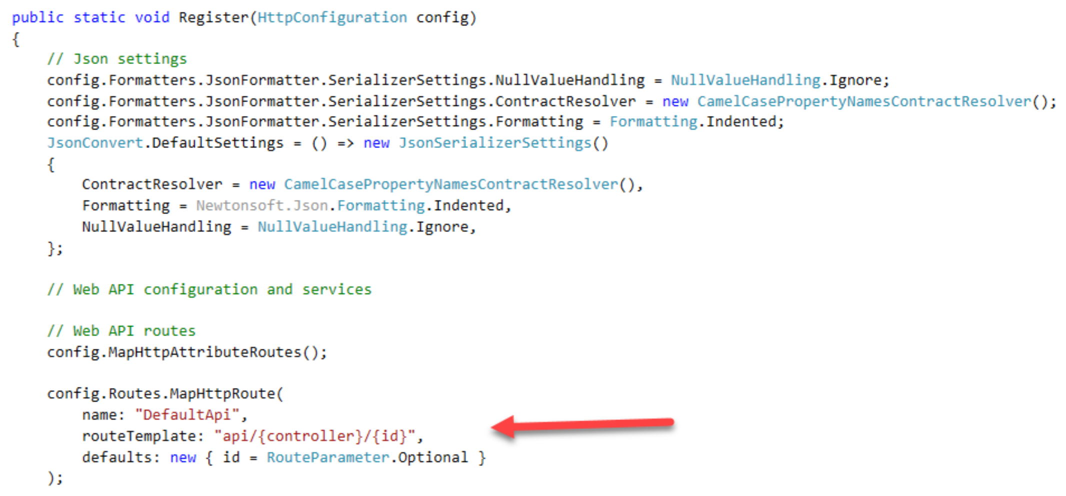
If you have used Web API previously, you will notice that the project that was set up is very similar to a WebApi project.

Detailed Steps



```
Solution 'DinnerBot' (1 project)
  DinnerBot
    Properties
    References
    App_Start
      WebApiConfig.cs
    Controllers
      MessagesController.cs
    default.htm
    Global.asax
    packages.config
    Web.config
```

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

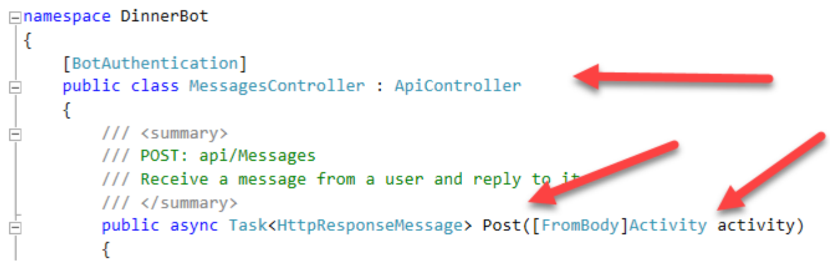


```
public static void Register(HttpConfiguration config)
{
    // Json settings
    config.Formatters.JsonFormatter.SerializerSettings.NullValueHandling = NullValueHandling.Ignore;
    config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();
    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

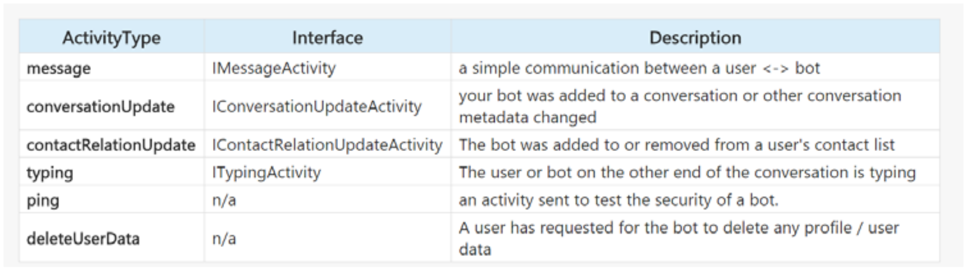
Detailed Steps



```
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)
        {
```

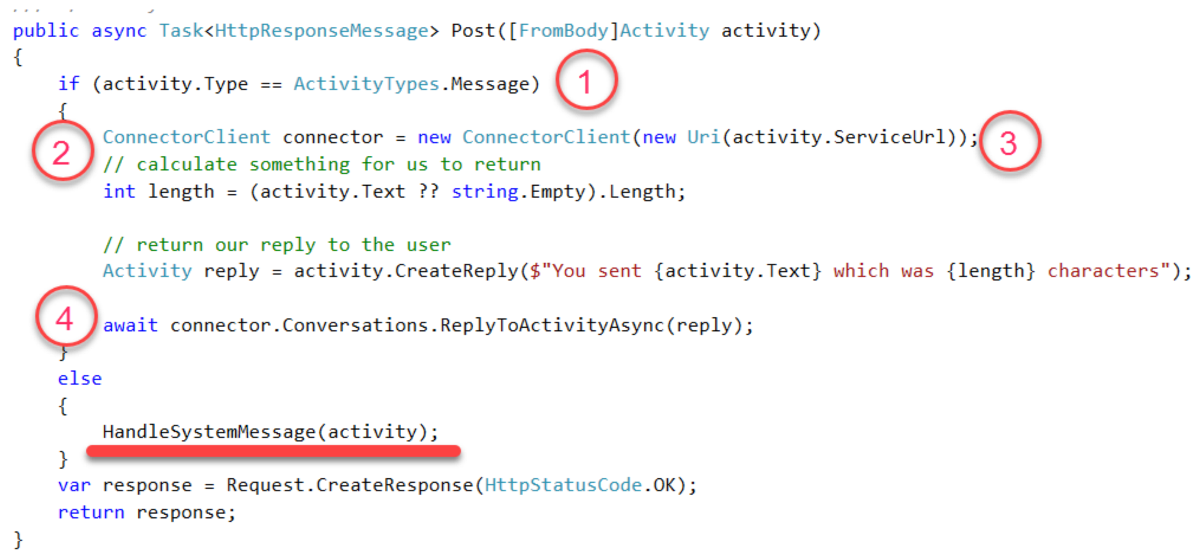
The first thing to notice is, as we discussed, it inherits from the ApiController . So any http Post to api/messages is routed to this method. Meaning all communication with your bot starts here. In addition, you can see it is being passed a type of Activity.

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

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



```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
        // calculate something for us to return
        int length = (activity.Text ?? string.Empty).Length;

        // return our reply to the user
        Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");

        await connector.Conversations.ReplyToActivityAsync(reply);
    }
    else
    {
        HandleSystemMessage(activity);
    }
    var response = Request.CreateResponse(HttpStatusCode.OK);
    return response;
}
```

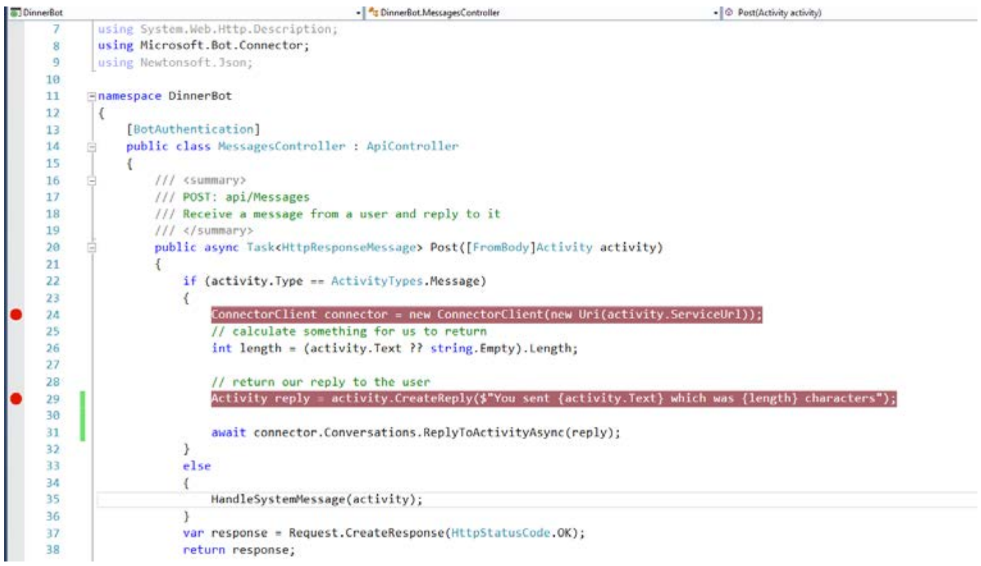
So once we know it's a **Message** (1). We create a **ConnectorClient** (2) and pass it a **ServiceURL** (3) All the rest of this sample is doing is reading the message and saying it back to the user with the length of the characters by using the **ReplyToActivityAsync** method (4) .

We will be making changes to this bot but first we need to make sure that we can test it using the

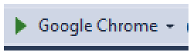
Detailed Steps

emulator. Make sure you have downloaded (https://docs.botframework.com/en-us/downloads/) and installed it before you begin.

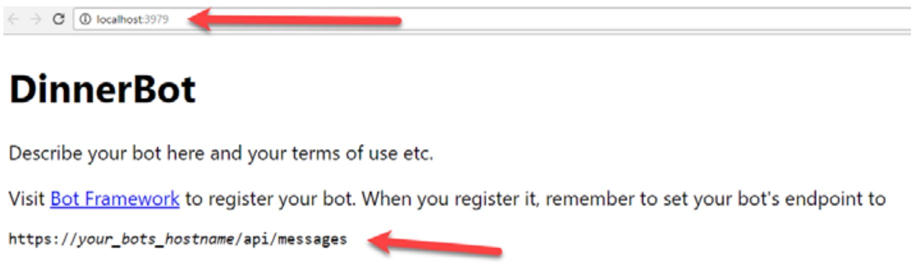
2. In Visual Studio, place a couple of breakpoints in the **MessagesController.cs** file so we can inspect things when we connect.



```
using System.Web.Http.Description;
using Microsoft.Bot.Connector;
using Newtonsoft.Json;

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)
            {
                ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
                // calculate something for us to return
                int length = (activity.Text ?? string.Empty).Length;

                // return our reply to the user
                Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");

                await connector.Conversations.ReplyToActivityAsync(reply);
            }
            else
            {
                HandleSystemMessage(activity);
            }
            var response = Request.CreateResponse(HttpStatusCode.OK);
            return response;
```

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.



DinnerBot

Describe your bot here and your terms of use etc.

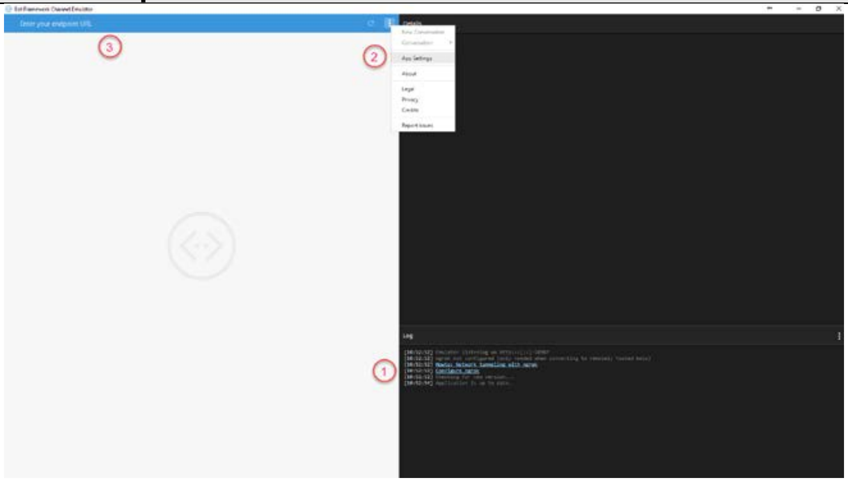
Visit Bot Framework to register your bot. When you register it, remember to set your bot's endpoint to

`https://your_bots_hostname/api/messages`

   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

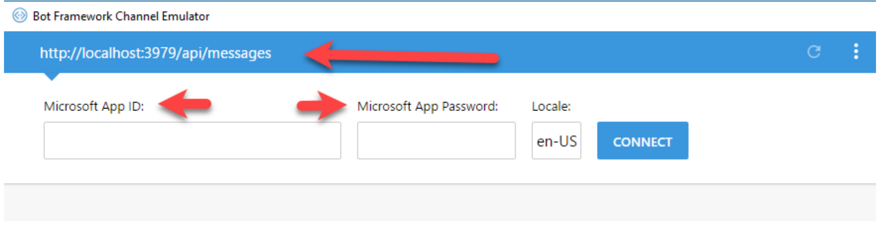


When it launches, you will notice a few things.
1) A log which shows the ServiceURL that the emulator is listening on, as well as a note to install NGrok which will be needed later for using the emulator with a cloud hosted bot.

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

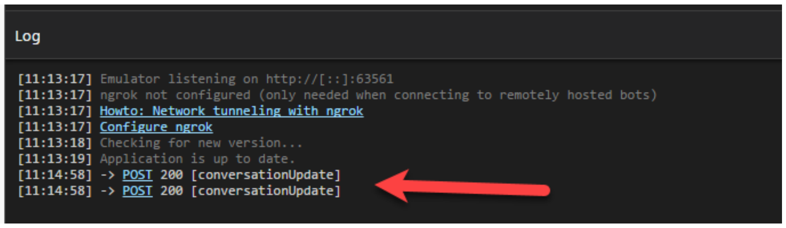
3) A prompt to enter the endpoint to your bot.

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)


Bot Framework Channel Emulator
http://localhost:3979/api/messages
Microsoft App ID:
Microsoft App Password:
Locale:
en-US
CONNECT

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] in your log

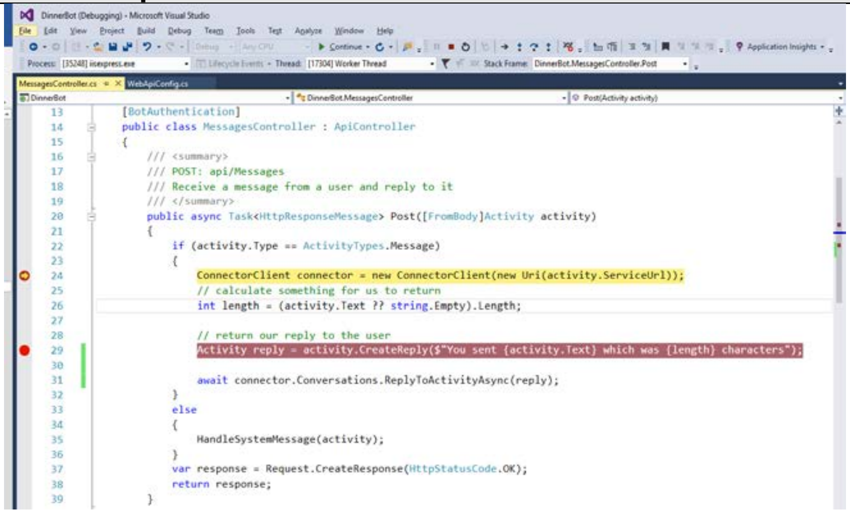


```
Log

[11:13:17] Emulator listening on http://[::]:63561
[11:13:17] ngrok not configured (only needed when connecting to remotely hosted bots)
[11:13:17] Howto: Network tunneling with ngrok
[11:13:17] Configure ngrok
[11:13:18] Checking for new version...
[11:13:19] Application is up to date.
[11:14:58] -> POST 200 [conversationUpdate]
[11:14:58] -> POST 200 [conversationUpdate]
```

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

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

Detailed Steps

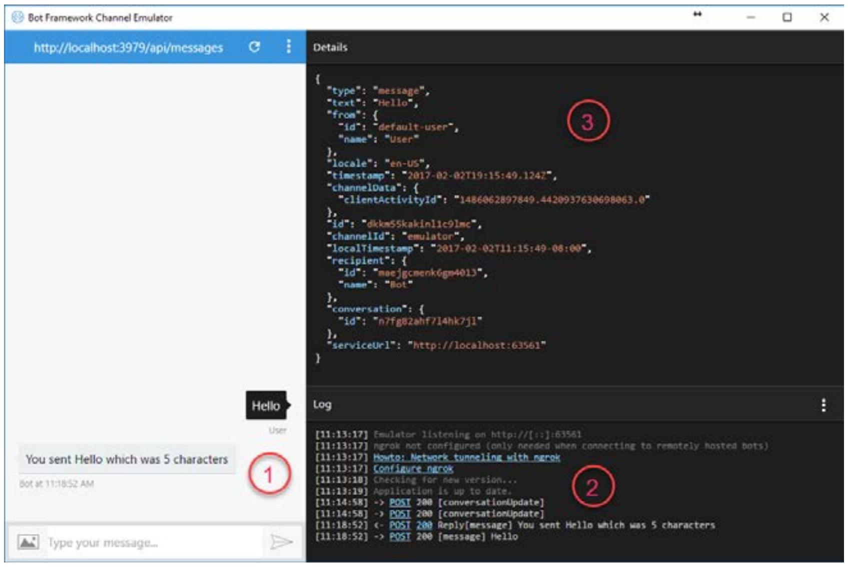

DinnerBot (Debugging) - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Continue
Application Insights
Process: [35248] iisexpress.exe
Lifecycle Events
Thread: [17304] Worker Thread
Stack Frame: DinnerBot.MessagesController.Post
MessagesController.cs
WebApiConfig.cs
DinnerBot
DinnerBot.MessagesController
Post(Activity activity)

```
[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)
        {
            ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
            // calculate something for us to return
            int length = (activity.Text ?? string.Empty).Length;

            // return our reply to the user
            Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");

            await connector.Conversations.ReplyToActivityAsync(reply);
        }
        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)


Bot Framework Channel Emulator
http://localhost:3979/api/messages
Details
3
Hello
User
You sent Hello which was 5 characters
Bot at 11:18:52 AM
1
Type your message...
Log
2

```
{
  "type": "message",
  "text": "Hello",
  "from": {
    "id": "default-user",
    "name": "User"
  },
  "locale": "en-US",
  "timestamp": "2017-02-02T19:15:49.124Z",
  "channelData": {
    "clientActivityId": "1486062897849.4420937630698063.0"
  },
  "id": "dkkm55kakinl1c9lmc",
  "channelId": "emulator",
  "localTimestamp": "2017-02-02T11:15:49-08:00",
  "recipient": {
    "id": "maejgcmenk6gm4013",
    "name": "Bot"
  },
  "conversation": {
    "id": "n7fg82ahf714hk7jl"
  },
  "serviceUrl": "http://localhost:63561"
}
```

```
[11:13:17] Emulator listening on http://[::]:63561
[11:13:17] ngrok not configured (only needed when connecting to remotely hosted bots)
[11:13:17] Howto: Network tunneling with ngrok
[11:13:17] Configure ngrok
[11:13:18] Checking for new version...
[11:13:19] Application is up to date.
[11:14:58] -> POST 200 [conversationUpdate]
[11:14:58] -> POST 200 [conversationUpdate]
[11:18:52] <- POST 200 Reply[message] You sent Hello which was 5 characters
[11:18:52] -> POST 200 [message] Hello
```

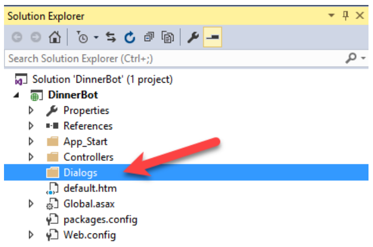
So in this section, we create a default hello world type of bot, got it up and running and interacted with in using the emulator. In the next section, we will start modifying it to create our dinner bot.

Exercise 2: Creating Dialogs

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

Detailed Steps

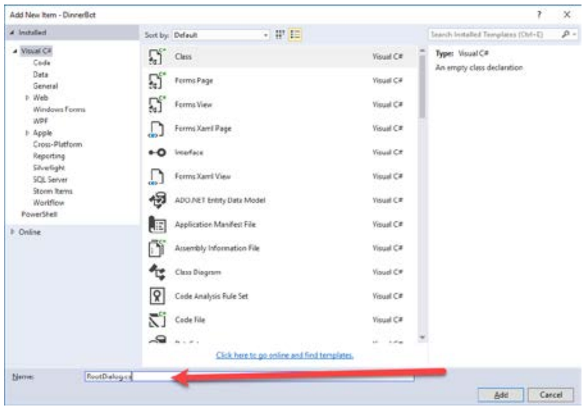
1. In your Solutions Explorer, right click on your project (DinnerBot) and select Add → New Folder



```
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'DinnerBot' (1 project)
  DinnerBot
    Properties
    References
    App_Start
    Controllers
    Dialogs
    default.htm
    Global.asax
    packages.config
    Web.config
```

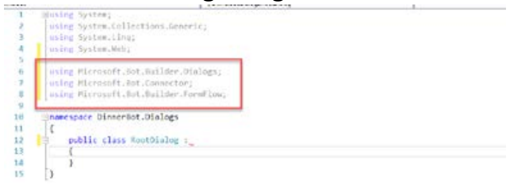
The first dialog we want to create is the RootDialog. This will be the place where all of our interaction flows.

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



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 **RootDialog.cs** file.



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

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

namespace DinnerBot.Dialogs
{
    public class RootDialog :
    {
    }
}
```

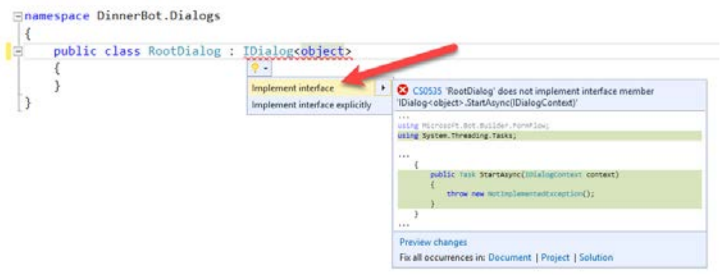
----- SNIP1-----------------------------------

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

Next, we need implement the IDialog Interface.

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

Detailed Steps



```
namespace DinnerBot.Dialogs
{
    public class RootDialog : IDialog<object>
    {
    }
}
```

Implement interface
Implement interface explicitly

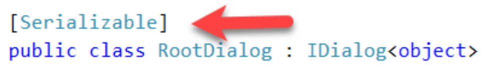
CS0535 'RootDialog' does not implement interface member 'IDialog<object>.StartAsync(IDialogContext)'

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

...
    {
        public Task StartAsync(IDialogContext context)
        {
            throw new NotImplementedException();
        }
    }
...
```

Preview changes
Fix all occurrences in: Document | Project | Solution

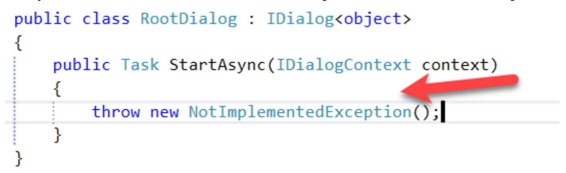
This will create a method called **StartAsync** which is what is called when we call the dialog.

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 RootDialog : IDialog<object>
```

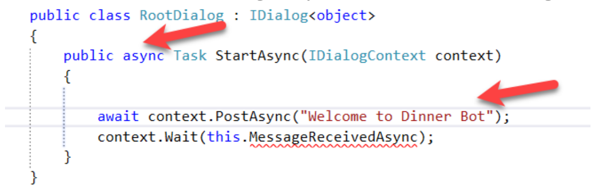
6. Replace the default **NotImplementedException**



```
public class RootDialog : IDialog<object>
{
    public Task StartAsync(IDialogContext context)
    {
        throw new NotImplementedException();
    }
}
```

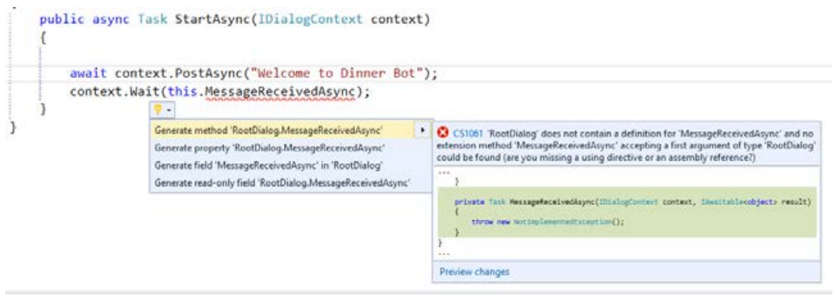
with the following code. Make sure you add the **async** keyword in front of Task in the method signature.

When this dialog is called, it will post back the message to the user. And then will wait for input form the user running any code in the **MessageRecievedAsync** method.



```
public class RootDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {

        await context.PostAsync("Welcome to Dinner Bot");
        context.Wait(this.MessageReceivedAsync);
    }
}
```

7. Next, we need to implement the MessageReceivedAsync method.



```
public async Task StartAsync(IDialogContext context)
{

    await context.PostAsync("Welcome to Dinner Bot");
    context.Wait(this.MessageReceivedAsync);
}
}
```

Generate method 'RootDialog.MessageReceivedAsync'
Generate property 'RootDialog.MessageReceivedAsync'
Generate field 'MessageReceivedAsync' in 'RootDialog'
Generate read-only field 'RootDialog.MessageReceivedAsync'

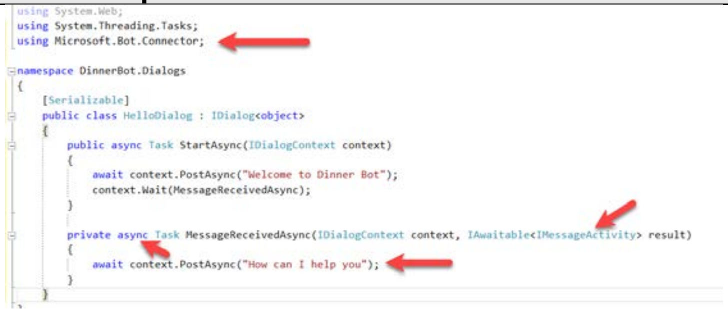
CS1061 'RootDialog' does not contain a definition for 'MessageReceivedAsync' and no extension method 'MessageReceivedAsync' accepting a first argument of type 'RootDialog' could be found (are you missing a using directive or an assembly reference?)

```
...
    }

    private Task MessageReceivedAsync(IDialogContext context, IAwaitable<object> result)
    {
        throw new NotImplementedException();
    }
}
...
```

Preview changes

for now, we are just going to post another message to the user. Add the following code to the **MessageReceivedAsync** method and add the **async** attribute.

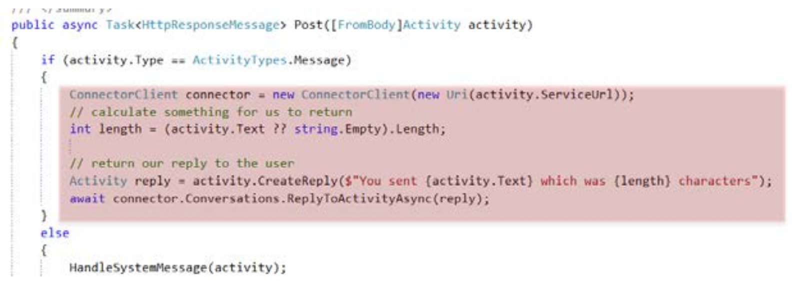
Detailed Steps



```
using System.Web;
using System.Threading.Tasks;
using Microsoft.Bot.Connector;

namespace DinnerBot.Dialogs
{
    [Serializable]
    public class HelloDialog : IDialog<object>
    {
        public async Task StartAsync(IDialogContext context)
        {
            await context.PostAsync("Welcome to Dinner Bot");
            context.Wait(MessageReceivedAsync);
        }

        private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
        {
            await context.PostAsync("How can I help you");
        }
    }
```

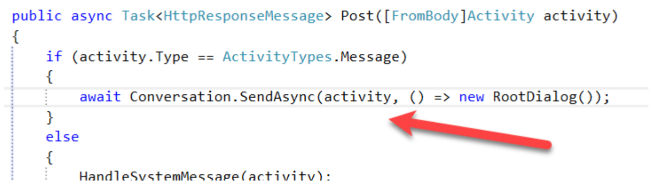
8. Add an **IMessageActivity** to the **IAwaitable**<> parameter. You will also need to add a **Microsoft.Bot.Connector** using statement as shown above.

   Now we need to have the bot find this dialog. For this we need to modify the **MessageController**
9. In the Solution Explorer open up the Controllers → **MessagesController.cs**
10. Remove the following code in the **ActivityType.Message** if statement.



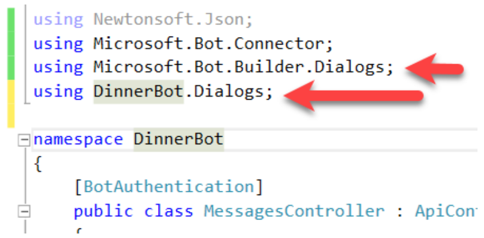
```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
        // calculate something for us to return
        int length = (activity.Text ?? string.Empty).Length;

        // return our reply to the user
        Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");
        await connector.Conversations.ReplyToActivityAsync(reply);
    }
    else
    {
        HandleSystemMessage(activity);
```

Replace with the following code. This tells the controller that if a message is received, route it to the **RootDialog**.



```
public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
{
    if (activity.Type == ActivityTypes.Message)
    {
        await Conversation.SendAsync(activity, () => new RootDialog());
    }
    else
    {
        HandleSystemMessage(activity):
```

Make sure you add the **Microsoft.Bot.Builder.Dialogs** and **DinnerBot.Dialogs** using statements to the top of the file.

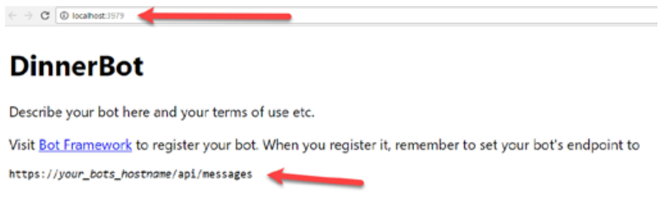


```
using Newtonsoft.Json;
using Microsoft.Bot.Connector;
using Microsoft.Bot.Builder.Dialogs;
using DinnerBot.Dialogs;

namespace DinnerBot
{
    [BotAuthentication]
    public class MessagesController : ApiCon
```

Let's test our new dialog.

Detailed Steps

11. Hit **F5** or press the green arrow ▶ Google Chrome ▾ to run your project. Make sure the browser launches.


localhost:3979

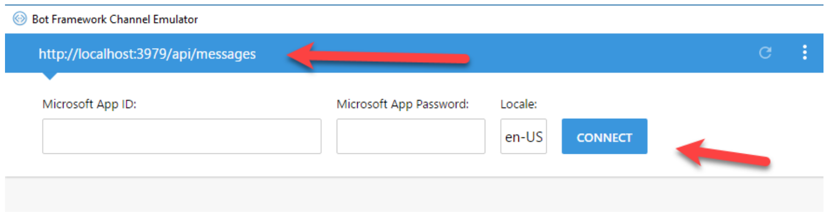
DinnerBot

Describe your bot here and your terms of use etc.

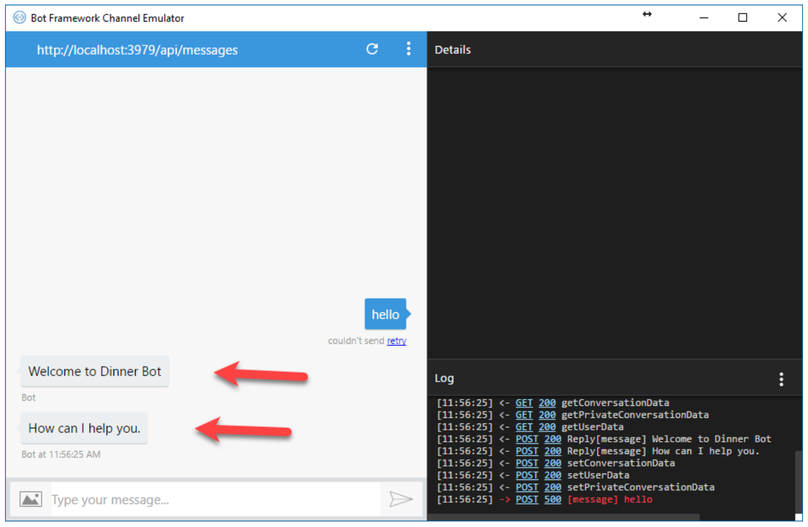
Visit Bot Framework to register your bot. When you register it, remember to set your bot's endpoint to

https://your_bots_hostname/api/messages

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


Bot Framework Channel Emulator
http://localhost:3979/api/messages
Microsoft App ID:
Microsoft App Password:
Locale:
en-US
CONNECT

Once the emulator launches, type in hello and the bot will now use our first dialog (the root dialog).



```
[11:56:25] <- GET 200 getConversationData
[11:56:25] <- GET 200 getPrivateConversationData
[11:56:25] <- GET 200 getUserData
[11:56:25] <- POST 200 Reply[message] Welcome to Dinner Bot
[11:56:25] <- POST 200 Reply[message] How can I help you.
[11:56:25] <- POST 200 setConversationData
[11:56:25] <- POST 200 setUserData
[11:56:25] <- POST 200 setPrivateConversationData
[11:56:25] -> POST 500 [message] hello
```

Now that we have a root dialog, let's do something besides just posting a simple message. We are going to give them an option to say hello or reserve a table.

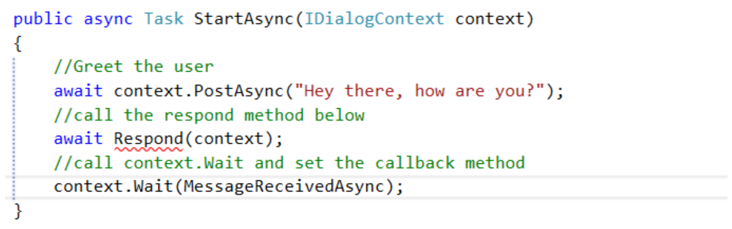
13. First we need to create a **HelloDialog**. Right Click on the dialogs folder and create **HelloDialog.cs** Making sure to:
Add the **Microsoft.Bot.Builder.Dialogs** using statement
Implement the **IDialog<>** interface,
Make the class **[Serializable]**

Detailed Steps

Add the **async** qualifier to the **StartAsync** method
(We will be pasting in the rest)
(For detailed instructions refer back to creating the **RootDialog** *above)*

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

14. Inside you **HelloDialog.cs** file, place the following code inside the StartAsync method.



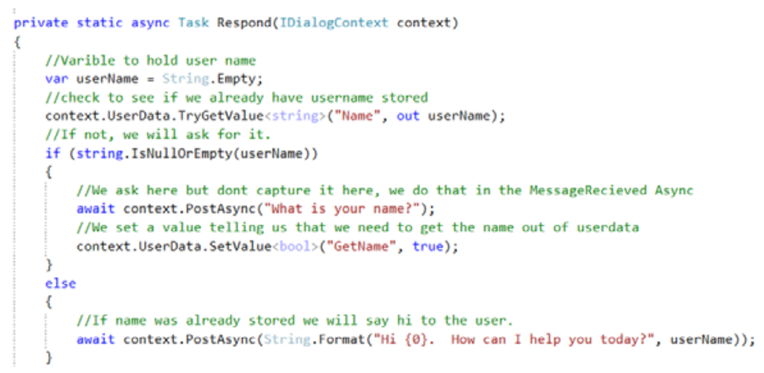
```
public async Task StartAsync(IDialogContext context)
{
    //Greet the user
    await context.PostAsync("Hey there, how are you?");
    //call the respond method below
    await Respond(context);
    //call context.Wait and set the callback method
    context.Wait(MessageReceivedAsync);
}
```

----- 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.

15. 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));
    }
```

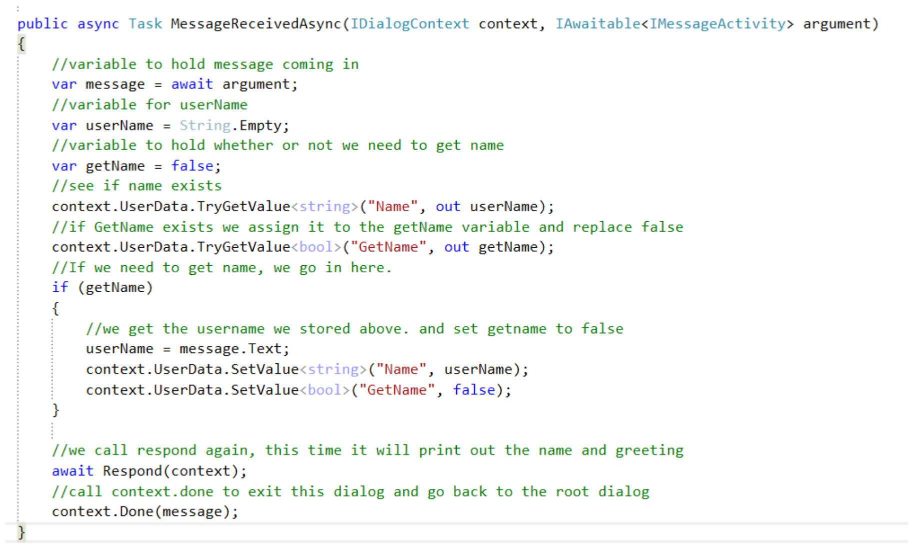
----- SNIP3----------------------------------

```
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
```

Detailed Steps

```
     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));
     }
 }
```

16. 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
```

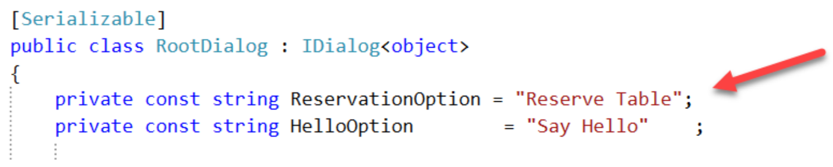
Detailed Steps

```
    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);
}
```

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** in order to send the user into the **HelloDialog**

17. 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"    ;
```

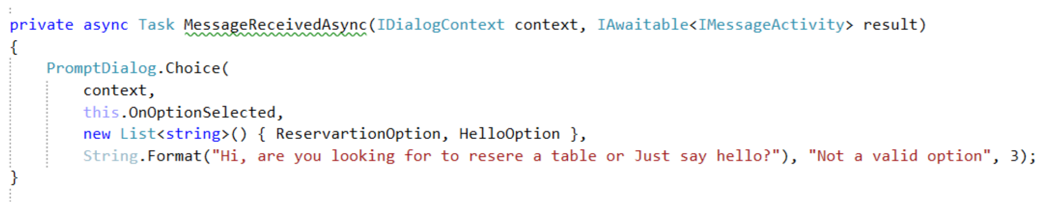
----- SNIP5-----------------------------------

```
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.

18. Paste the following code inside the **MessageReceivedAsync** method in the **RootDialog.cs** file. This will let them choose between reserving a table or just saying hello.



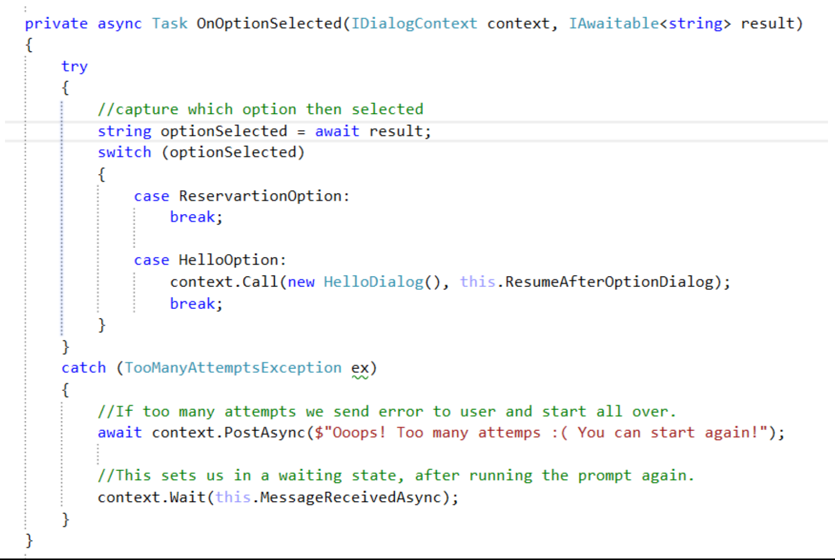
```
private async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> result)
{
    PromptDialog.Choice(
        context,
        this.OnOptionSelected,
        new List<string>() { ReservartionOption, HelloOption },
        String.Format("Hi, are you looking for to resere 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.

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



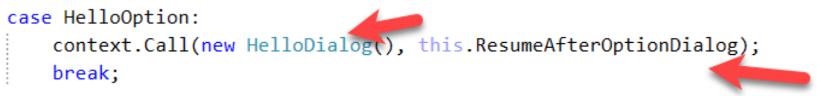
```
private async Task OnOptionSelected(IDialogContext context, IAwaitable<string> result)
{
    try
    {
        //capture which option then selected
        string optionSelected = await result;
        switch (optionSelected)
        {
            case ReservartionOption:
                break;

            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);
    }
}
```

Detailed Steps

----- 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);
    }
}
```

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**.

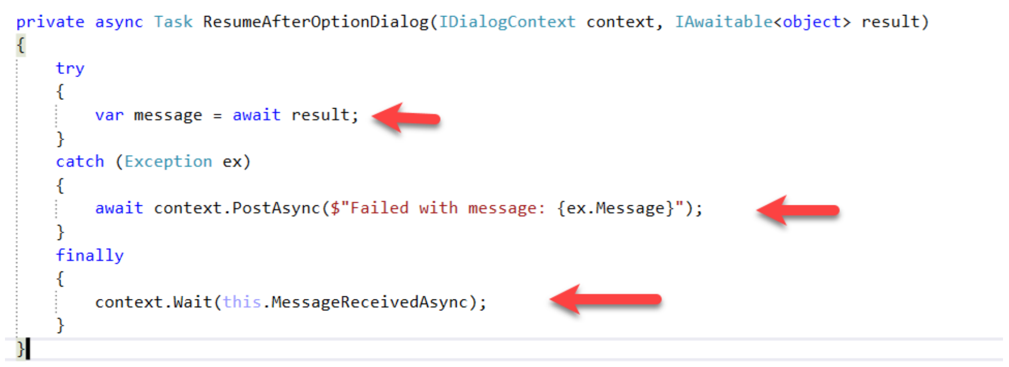


```
case HelloOption:
    context.Call(new HelloDialog(), this.ResumeAfterOptionDialog);
    break;
```

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.

20. 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.

Detailed Steps



```
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);
    }
}
```

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

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.

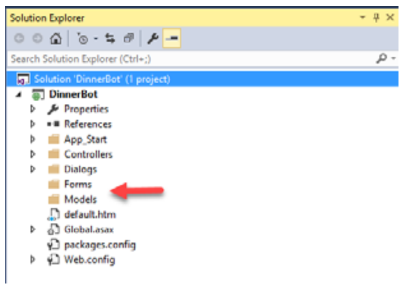
Exercise 3: Form Flow

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

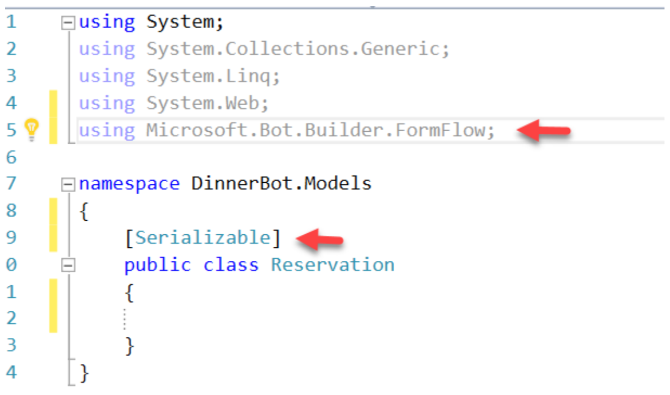
As we continue to work on the DinnerBot project, we will be modifying the project to work more closely with real world best practices. One of those, in the C# SDK, is the user of FormFlow. There are a few different ways to create FormFlows. We will utilize the separation of the model that the form flow 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 and create 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;



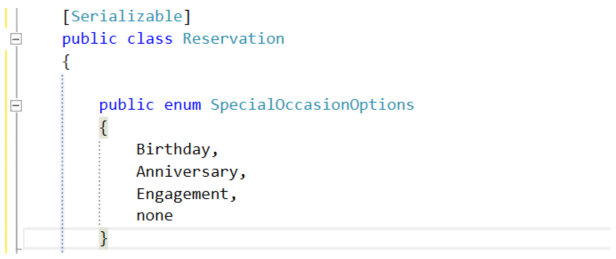
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.Bot.Builder.FormFlow;

namespace DinnerBot.Models
{
    [Serializable]
    public class Reservation
    {

    }
}
```

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

Detailed Steps

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----------------------------------

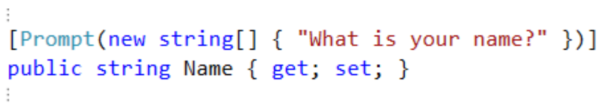
```
public enum SpecialOccasionOptions
{
   Birthday,
   Anniversary,
   Engagement,
   none
}
```

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----------------------------------

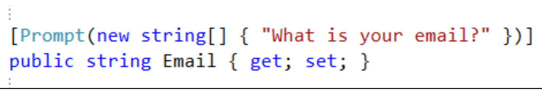
```
[Prompt(new string[] { "What is your name?" })]
public string Name { get; set; }

[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; }
```

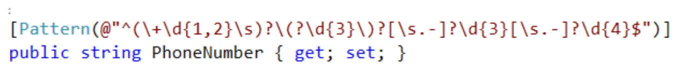
The second one is also a string to collect the email



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

Detailed Steps

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; }
```

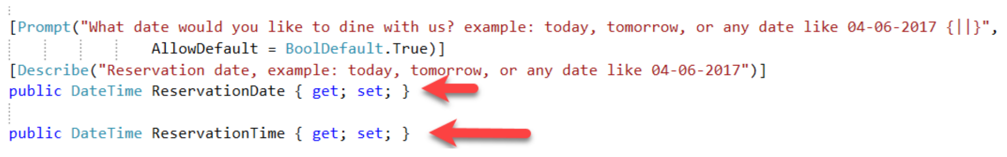
7. The next two properties will be for Reservation Date and Reservation Time. Paste them below the PhoneNumber property

----- 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.



```
[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; }
```

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;
public SpecialOccasionOptions? SpecialOccasion;

[Numeric(1, 5)]
[Optional]
```

Detailed Steps

```
[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----------------------------------

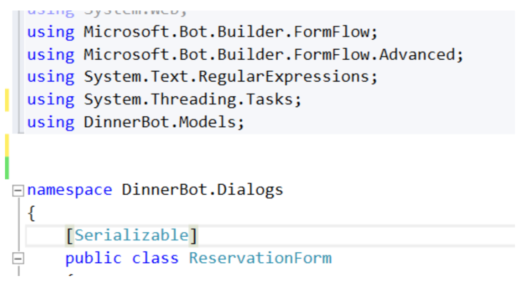
```
public Reservation(string name )
        {
            this.Name = name;
        }
```

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;



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

namespace DinnerBot.Dialogs
{
    [Serializable]
    public class ReservationForm
```

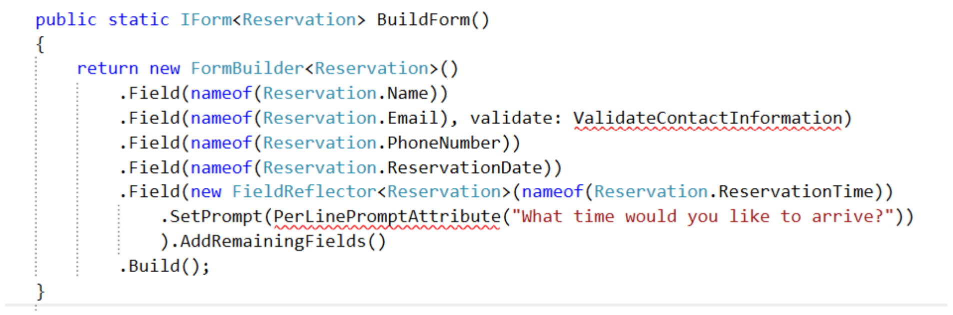
13. Inside the class, paste the following code.

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

Detailed Steps

```
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. 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();
    string contactInfo = string.Empty;
    if (GetEmailAddress((string)response, out contactInfo))
    {
        result.IsValid = true;
        result.Value = contactInfo;
    }
    else
```

Detailed Steps

```
    {
        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
    };
}
```

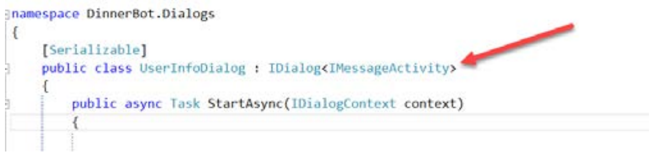
15. Now before we wire this up, we want to clean a few things up. The **HelloDialog** is doing more than just saying hello, it is also asking for a name and saving it. We want to abstract that out to its own dialog to hold User Info. Right-click on the Dialogs folder and **Add → Class** and call it **UserInfoDialog.cs**

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

Implement the **IDialog<IMessageActivity>** interface,
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 **RootDialog** *above)*

NOTE: Make sure the IDialog<> interface is using IMessageActivity and not Object

Detailed Steps



```
namespace DinnerBot.Dialogs
{
    [Serializable]
    public class UserInfoDialog : IDialog<IMessageActivity>
    {
        public async Task StartAsync(IDialogContext context)
        {
```

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);
    }
    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));

    }
}
```

Detailed Steps

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

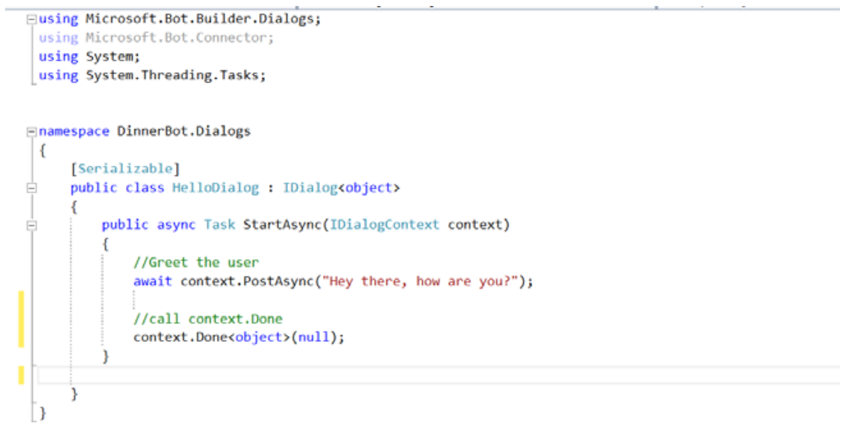
----- SNIP18-----------------------------------

```
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);
    }
    catch (Exception ex)
    {

        string message = ex.Message;
    }


}
```

Detailed Steps

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

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

    }
}
```

Detailed Steps

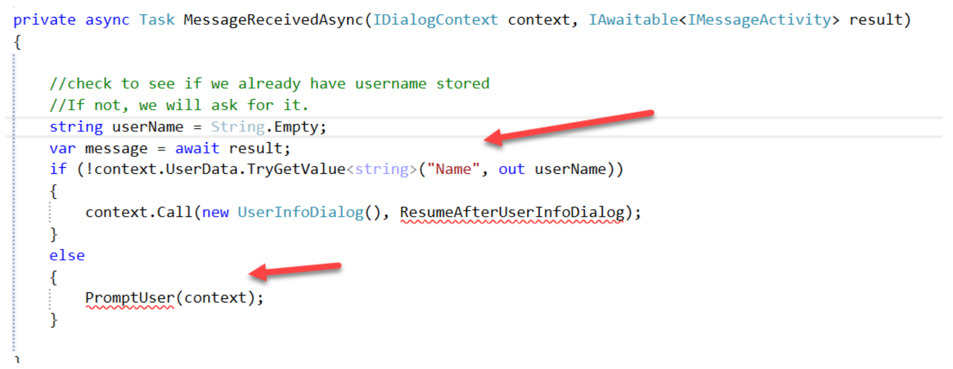
Now we want to go back to our Root Dialog and make some changes in order to call both our hello and our reservation dialogs. We want to set up some simple logic to check and see if we already know the name of the user and if not, call the **UserInfoDialog**.

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.



```
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----------------------------------

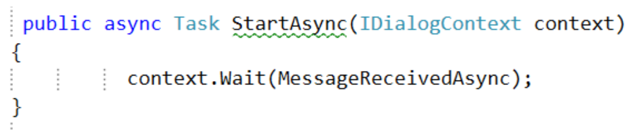
```
private void PromptUser(IDialogContext context)
{
```

Detailed Steps

```
    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. Delete the await **context.PostAsync("Welcome to Dinner Bot");** line from the **StartAsync** method so you are only left with **context.Wait(MessageReceivedAsync);**



```
public async Task StartAsync(IDialogContext context)
{
        context.Wait(MessageReceivedAsync);
}
```

22. 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.

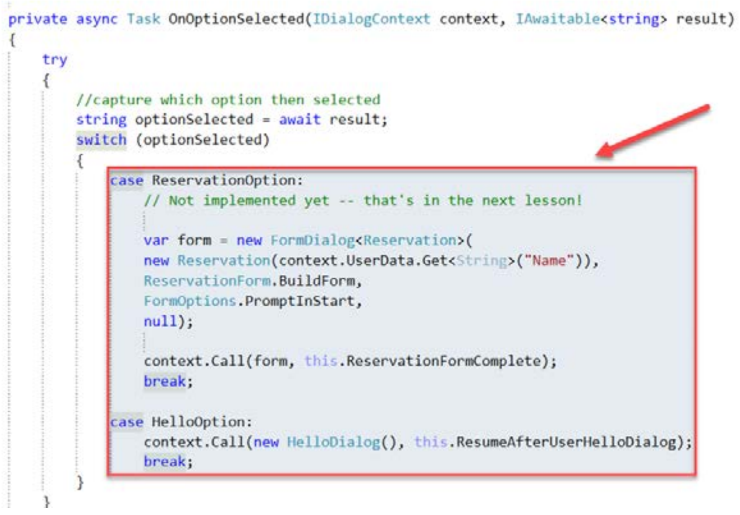
Now we want to update our **optionSelected** case statement inside of our **OnOptionSelected** method with the call to our **ReservationDialog.** We call this slightly differently since we are using Form Flow. In the context.Call, we pass it the Reservation with the name collected and saved in userData. 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).

23. 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----------------------------------

Detailed Steps

```
case ReservationOption:
    // Not implemented yet -- that's in the next lesson!

    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;
```



```
private async Task OnOptionSelected(IDialogContext context, IAwaitable<string> result)
{
    try
    {
        //capture which option then selected
        string optionSelected = await result;
        switch (optionSelected)
        {
            case ReservationOption:
                // Not implemented yet -- that's in the next lesson!

                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;
        }
    }
```

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

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.

24. 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-----------------------------------

Detailed Steps

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

25.

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

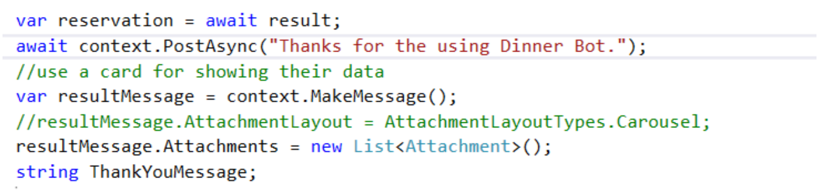
----- SNIP25-----------------------------------

```
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.";
        }
        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")),
```

Detailed Steps

```
            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.



```
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.

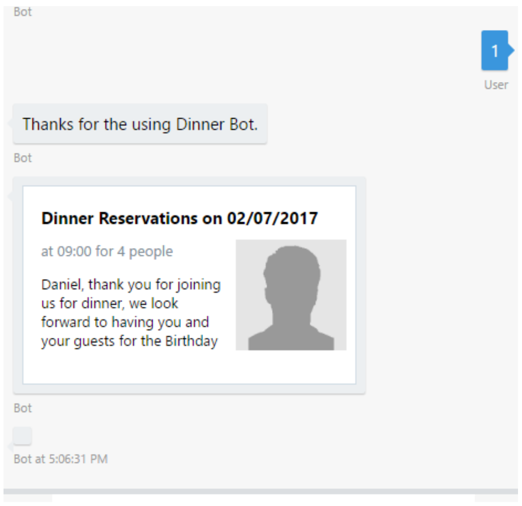
Detailed Steps

```
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.



Thanks for the using Dinner Bot.

Dinner Reservations on 02/07/2017
at 09:00 for 4 people
Daniel, thank you for joining us for dinner, we look forward to having you and your guests for the Birthday

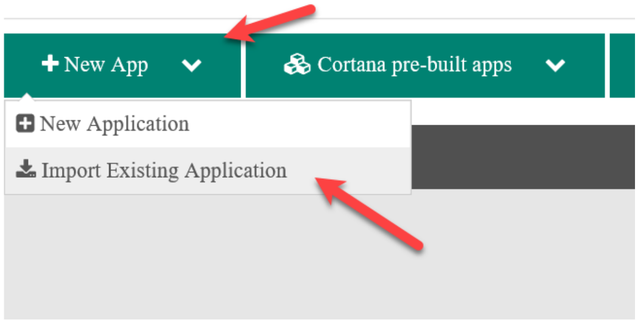
In the next exercise, we are going to ties all of this up to **LUIS** to get Natural Language Processing as part of your bot.

Exercise 4: Using Intent Dialogs (LUIS)

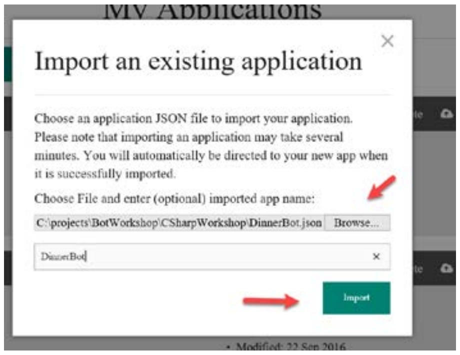
In this exercise we will import a LUIS Model that will handle questions coming from the users and route them to the appropriate Dialogs. We will not be creating the model but importing an already existing model. If you would like to learn how to create your own model you can find great tutorials and 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 the **New App → Import Existing Application**



3. Click browse 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.



My Applications

Import an existing application

Choose an application JSON file to import your application. Please note that importing an application may take several minutes. You will automatically be directed to your new app when it is successfully imported.

Choose File and enter (optional) imported app name:

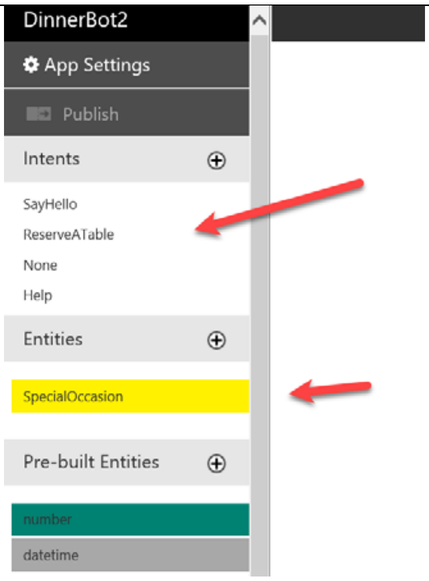
C:\projects\BotWorkshop\CSharpWorkshop\DinnerBot.json Browse...

DinnerBot

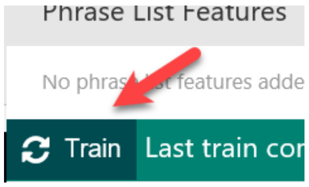
Import

• Modified: 22 Sep 2016

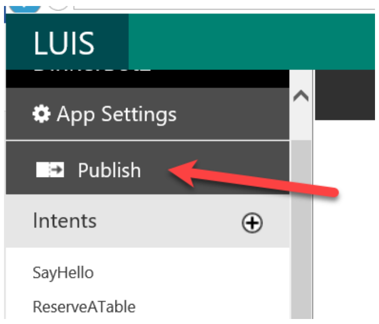
Once Imported you will notice that it has Intents for **SayHello**, **ReserveATable**, **None**, and **Help** and an entity of **SpecialOccasion**.



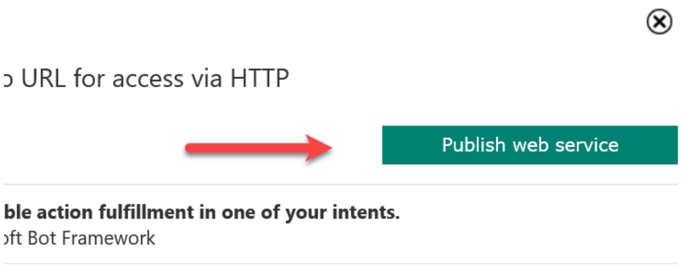
4. The next thing we need to do is train the model. Click the Train button on the bottom left.



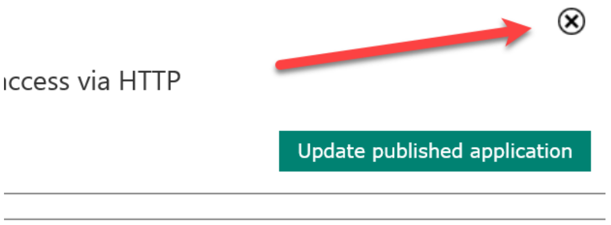
5. Once it is trained, we need to publish the model. In the upper left of the screen click on the Publish link.



When the window pops up, click on the Publish web service button. (Ignore the checkbox about the bot service)


ↄ URL for access via HTTP
Publish web service
ble action fulfillment in one of your intents.
ɔft Bot Framework

click on the x to close the window but leave LUIS open, we will need some information from here later on .



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

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

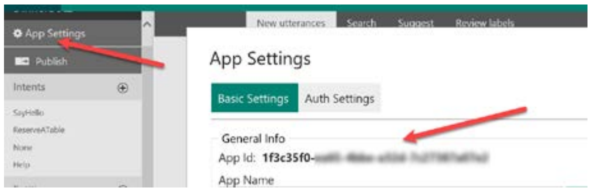
```
using Microsoft.Bot.Builder.Luis;
using Microsoft.Bot.Builder.Luis.Models;
```

7. 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>
    {
```

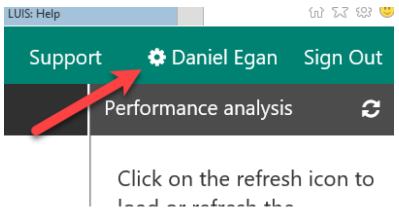
This will allow us to integrate with LUIS. We just need to add the **modelID** and Subscription key. We can get these from the LUIS.ai website.

8. Go back to the **LUIS.ai** website (Sign on if you need to) and open up your **DinnerBot** application. To get the **appID**. Click on the App Settings gear in the upper left corner. You will find **App Id** (the same thing as **modelID**) under Basic Settings



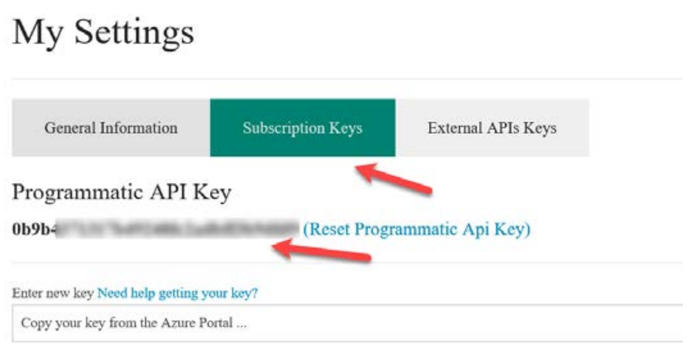
App Id: 1f3c35f0-[redacted]
App Name

9. To get your **Subscription key**, click on the gear icon next to your name on the LUIS.ai website.



Click on the refresh icon to

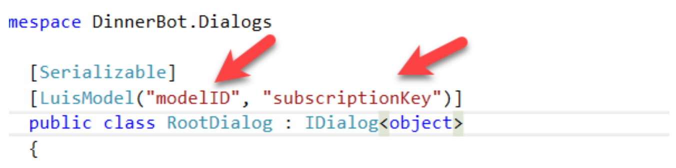
That will bring you to My Settings. You will find your subscription key under the Subscription Keys tab.



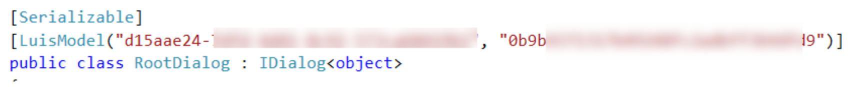
Programmatic API Key
0b9b4[redacted] (Reset Programmatic Api Key)

Enter new key Need help getting your key?
Copy your key from the Azure Portal ...

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)

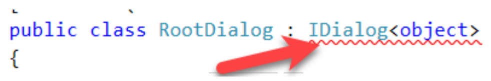


```
mespace DinnerBot.Dialogs

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

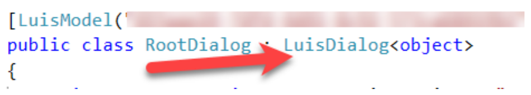


```
[Serializable]
[LuisModel("d15aae24-[redacted]", "0b9b[redacted]19")]
public class RootDialog : IDialog<object>
```

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

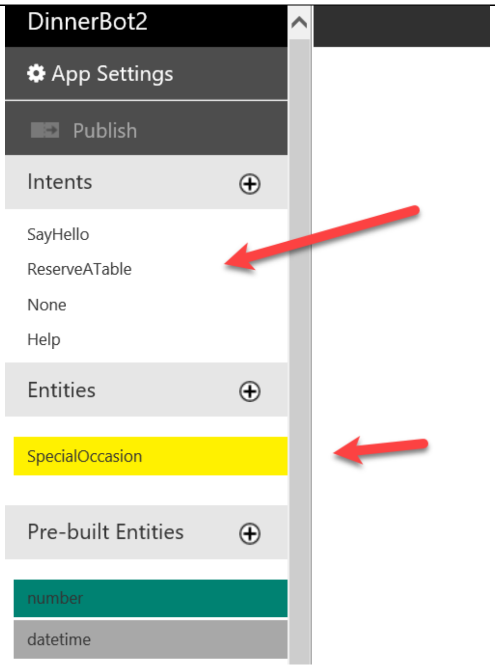


```
 public class RootDialog : IDialog<object>
 {
```



```
[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.

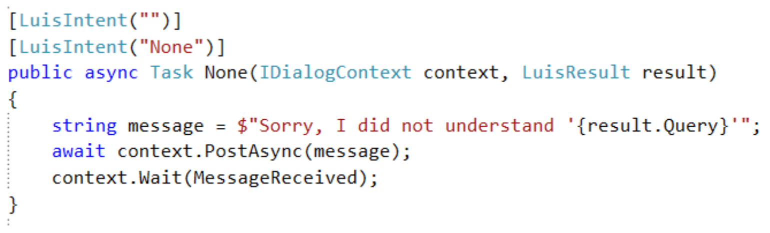
This code **REPLACES** the **StartAsync** method in RootDialog. We don't need it since we are not implementing IDialog<>

------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.");
```

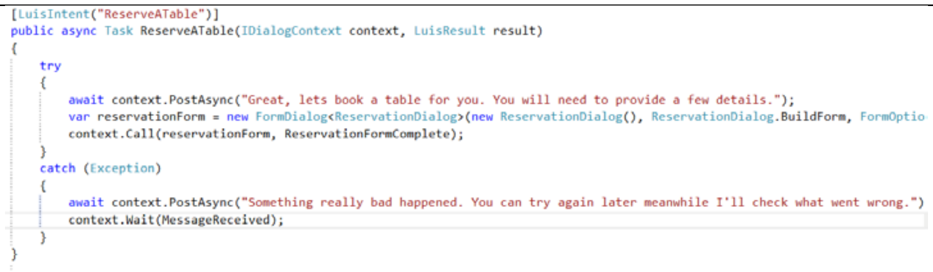
```
        var reservationForm = new FormDialog<ReservationDialog>(new
ReservationDialog(), ReservationDialog.BuildForm, FormOptions.PromptInStart);
        context.Call(reservationForm, ReservationFormComplete);
    }
    catch (Exception)
    {
        await context.PostAsync("Something really bad happened. You can try again
later meanwhile I'll check what went wrong.");
        context.Wait(MessageReceived);
    }
}

[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.



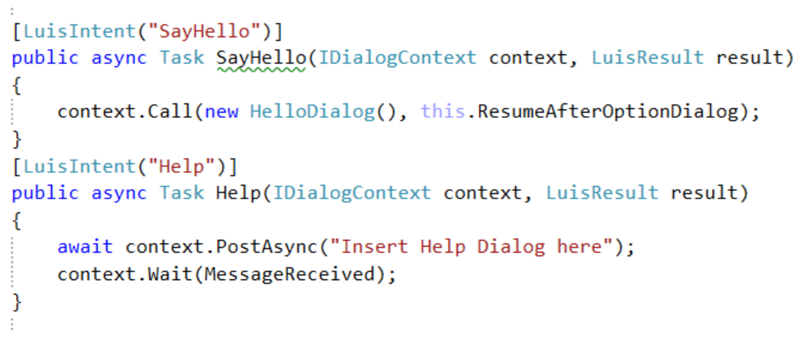
```
[LuisIntent("")]
[LuisIntent("None")]
public async Task None(IDialogContext context, LuisResult result)
{
    string message = $"Sorry, I did not understand '{result.Query}'";
    await context.PostAsync(message);
    context.Wait(MessageReceived);
}
```

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