

+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.....	26
Exercise 4: Using Intent Dialogs (LUIS).....	34
Additional Resources .....	41
Copyright .....	42

---

# 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

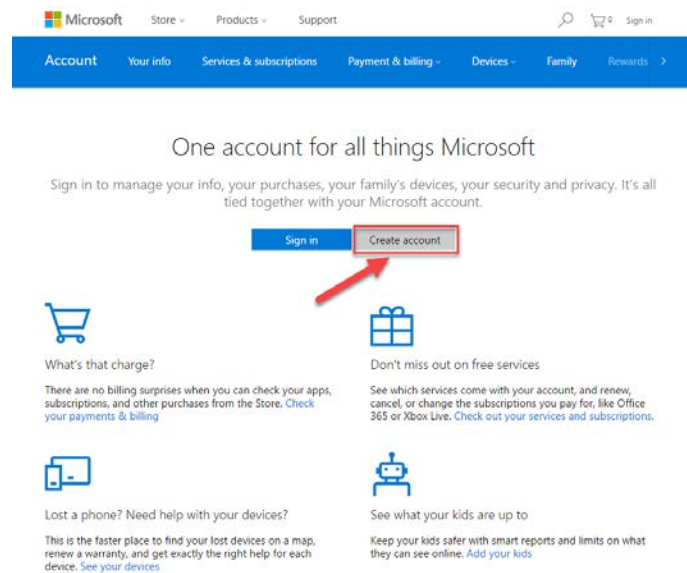
This series of exercises is designed to show you how to get started using the Microsoft Bot Framework. In this lab, we are going to create a DinnerBot that will allow you to make reservations for a restaurant.

### 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 C:\Users\%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**.



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

# Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

Page 5 of 42

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.

Next

Microsoft

**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](#)

Resend email

[Terms of Use](#) [Privacy & Cookies](#) [Sign out](#)

Microsoft

Google

Gmail +

COMPOSE

Inbox (1)

Starred

Sent Mail

Drafts

More

Daniel

Verify your email address

Microsoft account team - account 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.

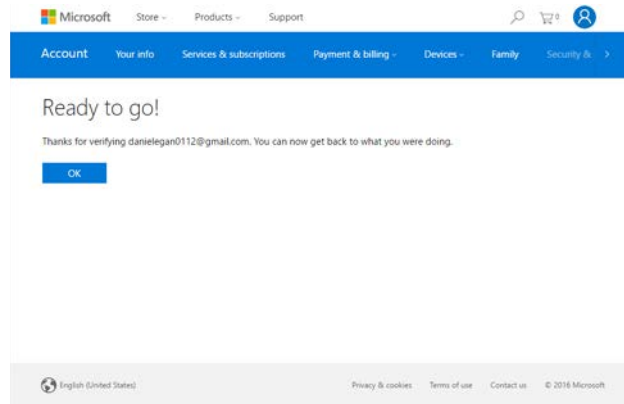
Thanks,

The Microsoft account team

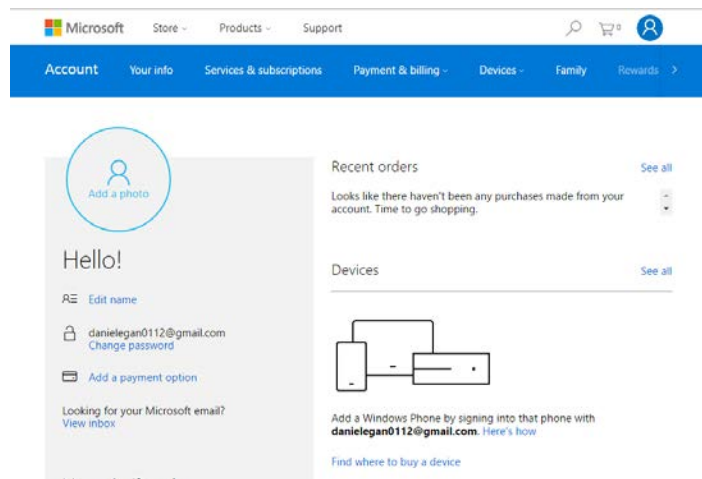
# Creating a bot using the Microsoft Bot Framework

## C# Hands-on Labs

Page 6 of 42

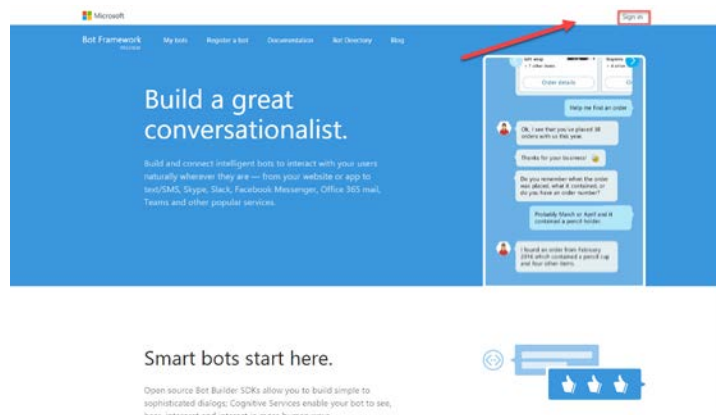


- Either path will take you to this screen



## 2. Create a BotFramework account

- Navigate to <http://BotFramework.com>
- Click on sign in



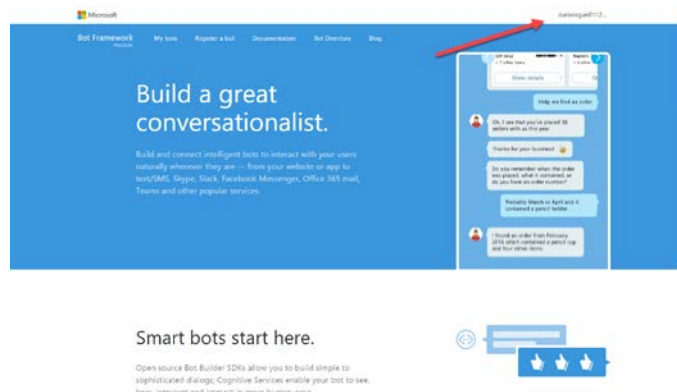
- If you are using the same browser that you used to create your Microsoft ID then you will be signed in automatically, otherwise you will need to use the

## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

Page 7 of 42

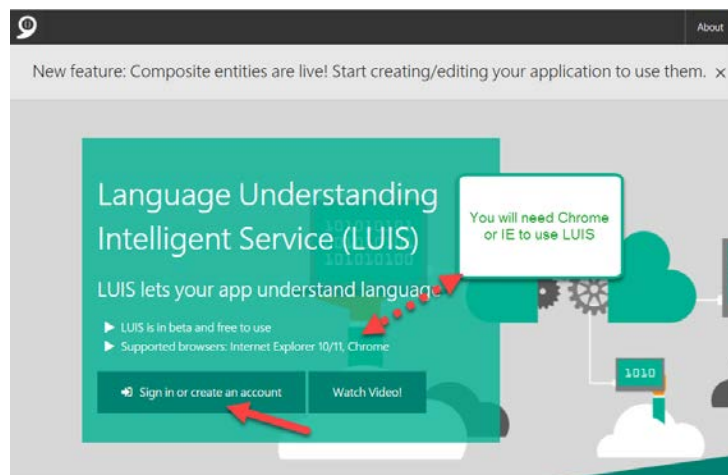
ID you just created to sign in.



- 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

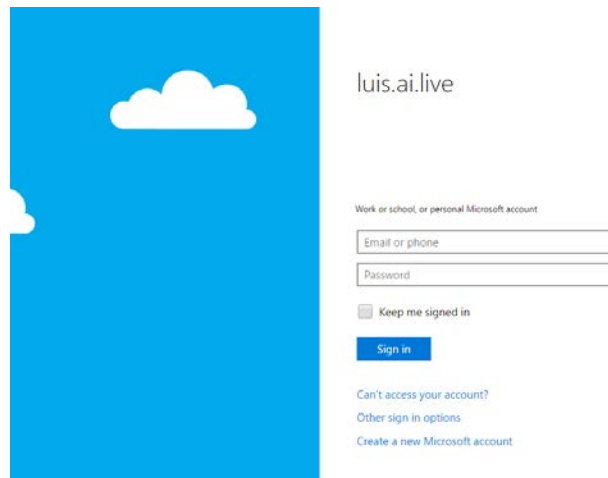


- Sign in with your Microsoft account

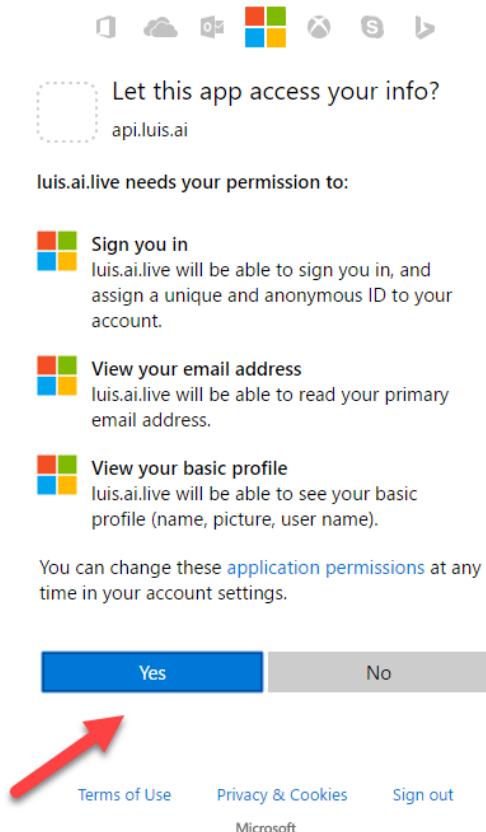
## Creating a bot using the Microsoft Bot Framework

C# Hands-on Labs

Page 8 of 42



- 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.



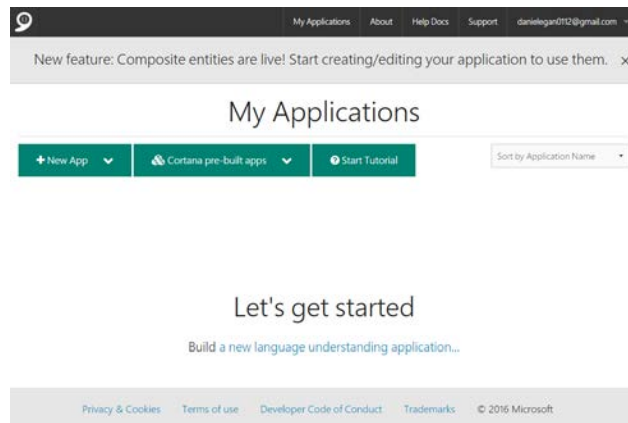
- You can walk through the quick walkthrough if you would like or click the x to close it. When finished, your screen should look like below.



# Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

Page 9 of 42



- 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.

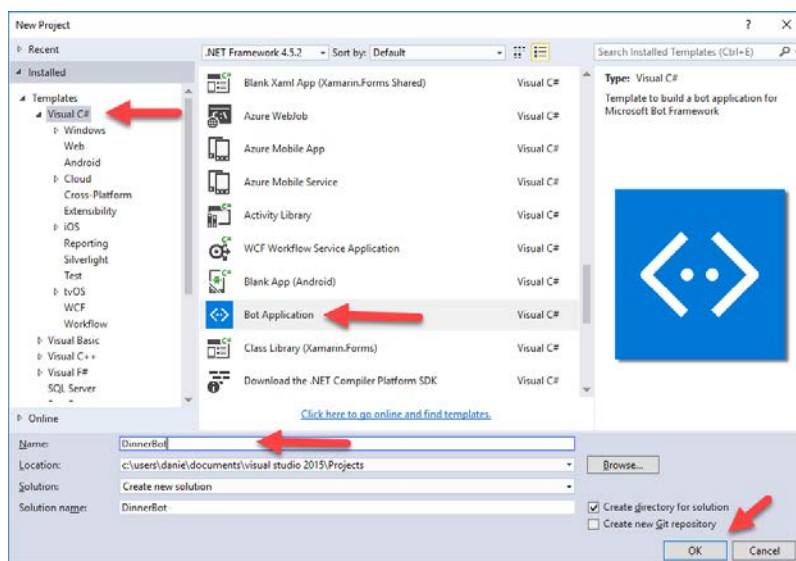
# Exercise 1: Basic Bot using BotBuilder

In this exercise, you will create a simple bot using the BotBuilder and

## 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



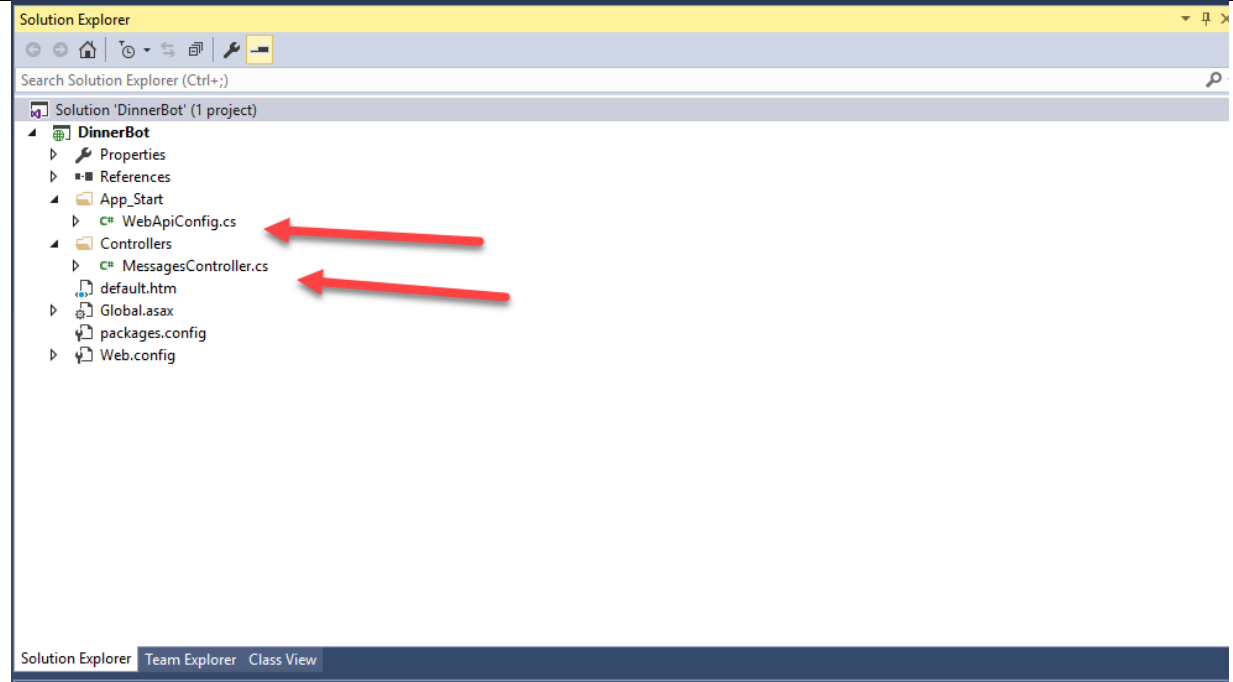
If you have used Web API previously, you will notice that the project that was set up is very similar to a WebApi project.

## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

Page 11 of 42

### Detailed Steps



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

## Creating a bot using the Microsoft Bot Framework

### C# Hands-on Labs

Page 12 of 42

#### 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	ICConversationUpdateActivity	your bot was added to a conversation or other conversation metadata changed
contactRelationUpdate	ICContactRelationUpdateActivity	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

## Creating a bot using the Microsoft Bot Framework

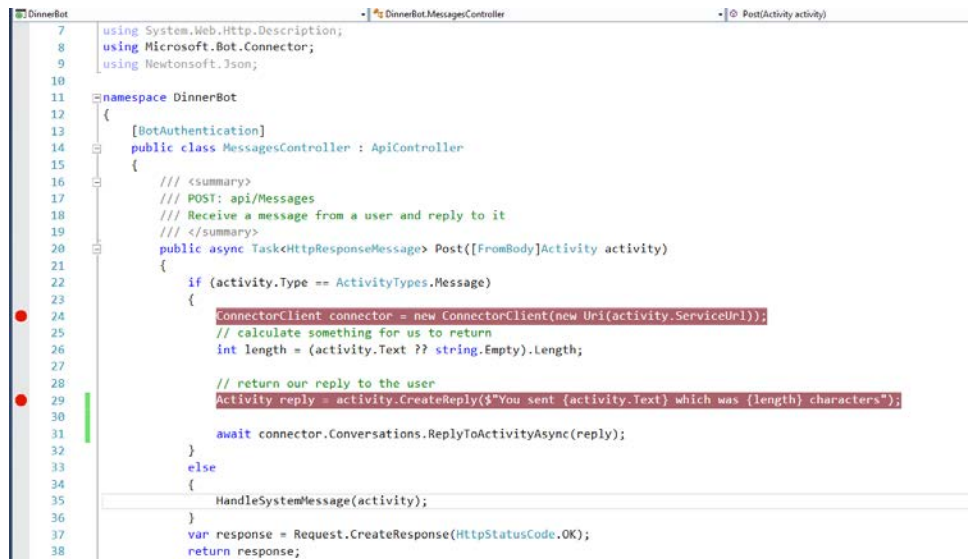
Node.js Hands-on Labs

Page 13 of 42

### Detailed Steps

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

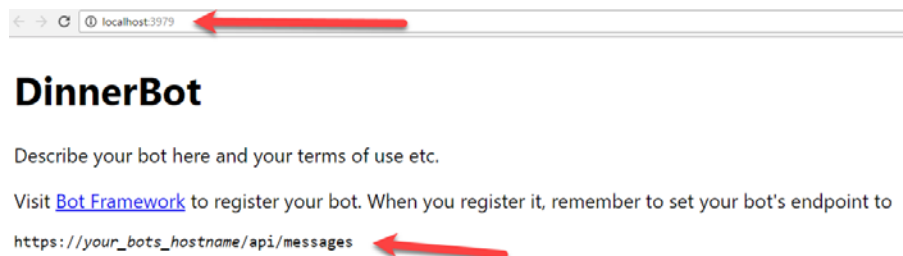
2. In Visual Studio, place a couple of breakpoints in the **MessagesController.cs** file so we can inspect things when we connect.



```
7 using System.Web.Http.Description;
8 using Microsoft.Bot.Connector;
9 using Newtonsoft.Json;
10
11 namespace DinnerBot
12 {
13     [BotAuthentication]
14     public class MessagesController : ApiController
15     {
16         /// <summary>
17         /// POST: api/Messages
18         /// Receive a message from a user and reply to it
19         /// </summary>
20         public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
21         {
22             if (activity.Type == ActivityTypes.Message)
23             {
24                 ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
25                 // calculate something for us to return
26                 int length = (activity.Text ?? string.Empty).Length;
27
28                 // return our reply to the user
29                 Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");
30
31                 await connector.Conversations.ReplyToActivityAsync(reply);
32             }
33             else
34             {
35                 HandleSystemMessage(activity);
36             }
37             var response = Request.CreateResponse(HttpStatusCode.OK);
38             return response;
39         }
40     }
41 }
```

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.

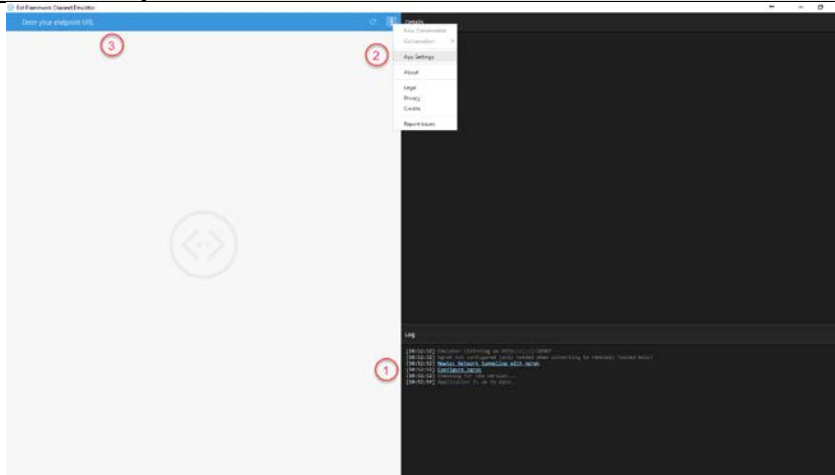
4. Run the Bot Framework Channel Emulator that you previously installed.

## Creating a bot using the Microsoft Bot Framework

C# Hands-on Labs

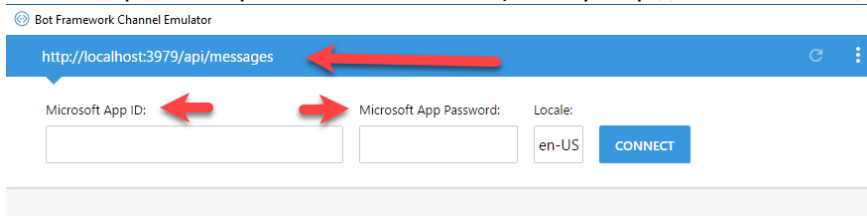
Page 14 of 42

### Detailed Steps



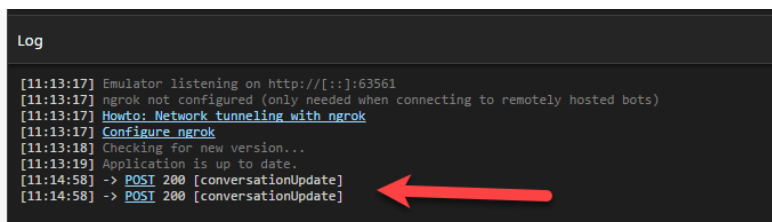
When it launches, you will notice a few things.

- 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 [ConversationUpdate] in your log



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

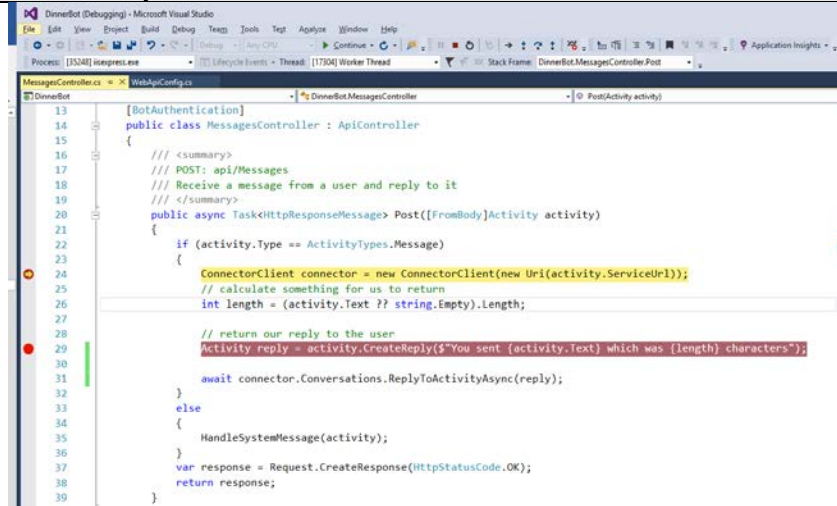
Once you hit enter, you should hit the breakpoints you set in Visual Studio.

## Creating a bot using the Microsoft Bot Framework

### Node.js Hands-on Labs

Page 15 of 42

#### Detailed Steps

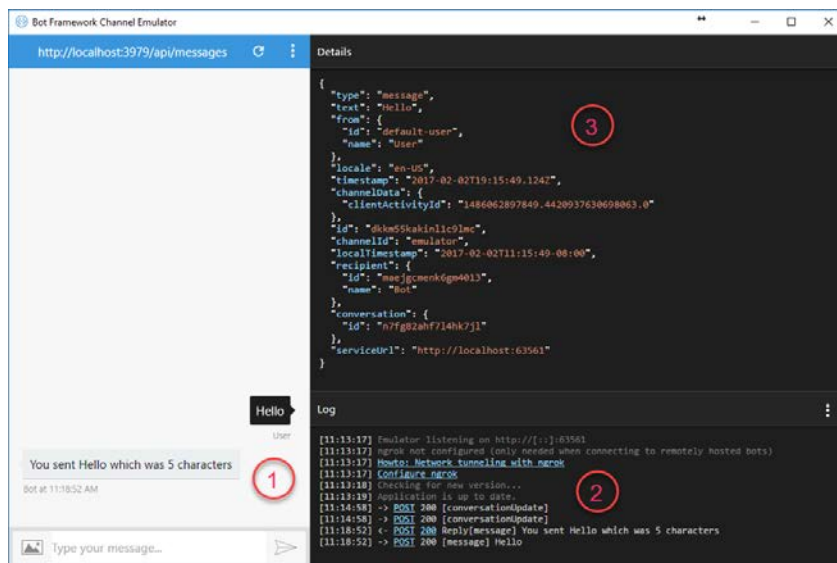


```
13 [BotAuthentication]
14 public class MessagesController : ApiController
15 {
16     /// <summary>
17     /// POST: api/Messages
18     /// Receive a message from a user and reply to it
19     /// </summary>
20     public async Task<HttpResponseMessage> Post([FromBody]Activity activity)
21     {
22         if (activity.Type == ActivityTypes.Message)
23         {
24             ConnectorClient connector = new ConnectorClient(new Uri(activity.ServiceUrl));
25             // calculate something for us to return
26             int length = (activity.Text ?? string.Empty).Length;
27
28             // return our reply to the user
29             Activity reply = activity.CreateReply($"You sent {activity.Text} which was {length} characters");
30             await connector.Conversations.ReplyToActivityAsync(reply);
31         }
32         else
33         {
34             HandleSystemMessage(activity);
35         }
36         var response = Request.CreateResponse(HttpStatusCode.OK);
37         return response;
38     }
39 }
```

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 breakpoints and it **F5** to continue.

If you return back to the emulator, you will see the response 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)



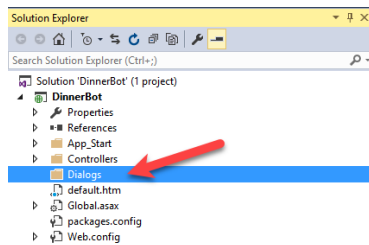
So in this section, we create a default hello world type of bot, got it up and running and interacted with in using the emulator. In the next section, we will start modifying it to create our dinner bot.

## Exercise 2: Creating Dialogs

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

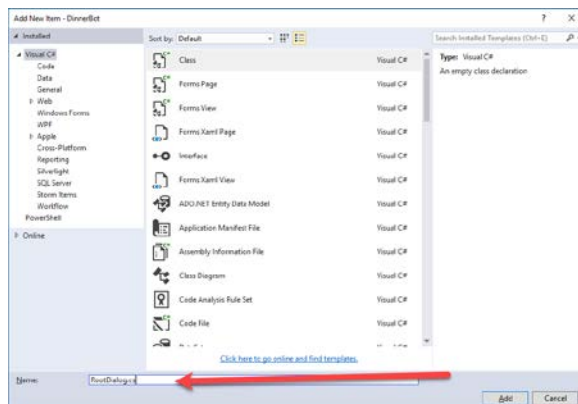
### Detailed Steps

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



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.

```
1 using System;  
2 using System.Collections.Generic;  
3 using System.Linq;  
4 using System.Web;  
5  
6 using Microsoft.Bot.Builder.Dialogs;  
7 using Microsoft.Bot.Connector;  
8 using Microsoft.Bot.Builder.FormFlow;  
9  
10 namespace DinnerBot.Dialogs  
11 {  
12     public class RootDialog :  
13     {  
14     }  
15 }
```

----- 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



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

- 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>
    
```

- Replace the default **NotImplementedException**

```

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

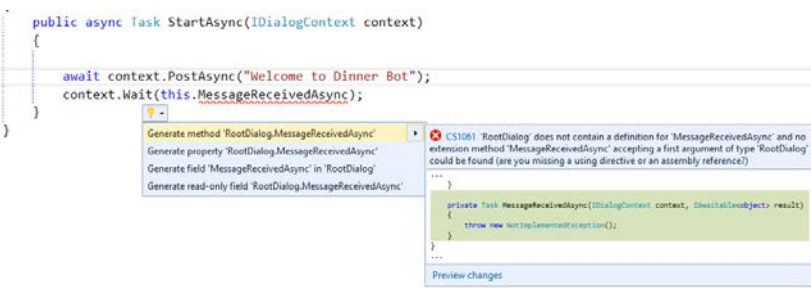
with the following code. Make sure you add the **async** keyword in front of Task in the method signature.

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

```

public class RootDialog : IDialog<object>
{
    public async Task StartAsync(IDialogContext context)
    {
        await context.PostAsync("Welcome to Dinner Bot");
        context.Wait(this.MessageReceivedAsync);
    }
}
    
```

- Next, we need to implement the MessageReceivedAsync method.



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.

## Creating a bot using the Microsoft Bot Framework

### C# Hands-on Labs

Page 18 of 42

#### 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);
    }
}
```

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 : ApiController
    {
    }
}
```

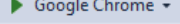
Let's test our new dialog.

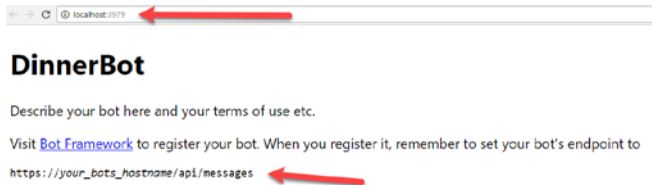
## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

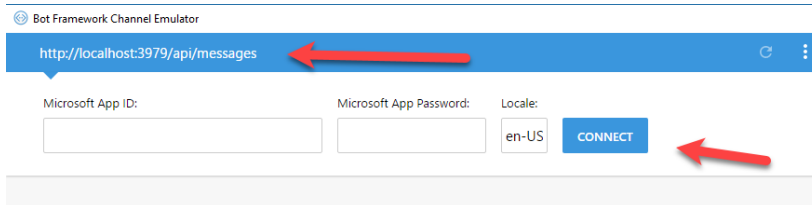
Page 19 of 42

### Detailed Steps

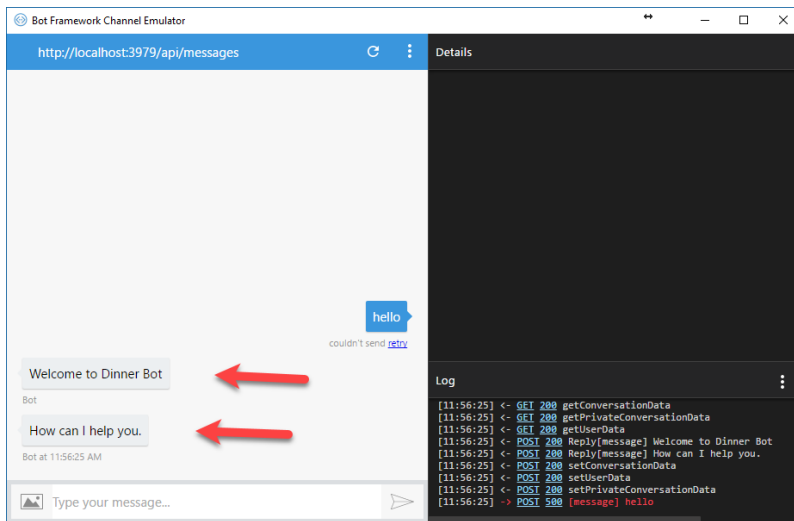
11. Hit **F5** or press the green arrow  to run your project. Make sure the browser launches.



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



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



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:  
Implement the **IDialog<>** interface,  
Add the **Microsoft.Bot.Builder.Dialogs** using statement  
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)
{
    //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 MessageReceived 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
    MessageReceived 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 ReservartionOption = "Reserve Table";
    private const string HelloOption = "Say Hello";
}
```



----- SNIP5-----

```
private const string ReservartionOption = "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>() { ReservationOption, 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 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

----- SNIP7-----

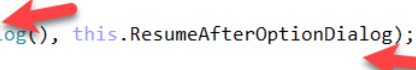
```
private async Task OnOptionSelected(IDialogContext context,
IAwaitable<string> result)
{
    try
    {
        //capture which option then selected
        string optionSelected = await result;
        switch (optionSelected)
        {
            case ReservationOption:
                break;

            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.



## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

Page 25 of 42

### 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.

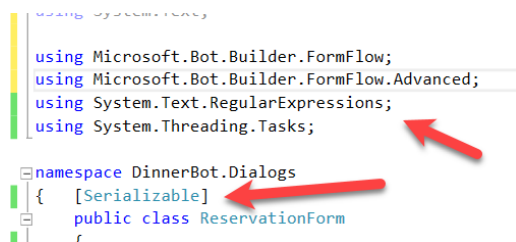
## Exercise 3: Form Flow

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

21. Open up Visual Studio and in the Solution Explorer, right click on Dialogs and create class called **ReservationDialog.cs**.
22. Add the **[Serializable]** attribute to the top of the class.
23. Add the following Using Statements to the top of the class.

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

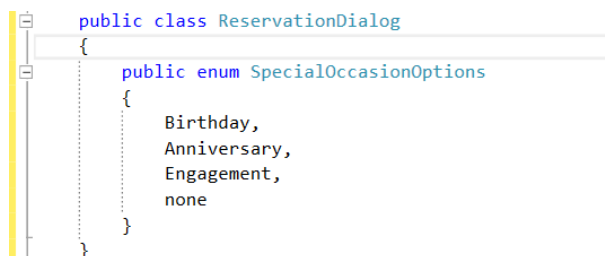


```
using System.Linq;  
  
using Microsoft.Bot.Builder.FormFlow;  
using Microsoft.Bot.Builder.FormFlow.Advanced;  
using System.Text.RegularExpressions;  
using System.Threading.Tasks;  
  
namespace DinnerBot.Dialogs  
{  
    [Serializable]  
    public class ReservationForm  
    {  
    }  
}
```

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.

24. 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.



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

----- SNIP8-----

```
public enum SpecialOccasionOptions  
{  
    Birthday,  
    Anniversary,
```

### Detailed Steps

```
Engagement,
none
}
```

25. 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.

----- SNIP9-----

```
[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}$")]
```

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; }
:

```

The third one is a bit different, it uses a [Patter] 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; }
```

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

----- SNIP9-----

```
[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; }
```

### Detailed Steps

```
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; }
```

27. 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.

----- SNIP10-----

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

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

28. Now we need to create the build form for the Dialog. Paste the following code below the Rating Property

----- SNIP11-----

```
public static IForm<ReservationDialog> BuildForm()
{
    return new FormBuilder<ReservationDialog>()
        .Field(nameof(Name))
        .Field(nameof(Email), validate: ValidateContactInformation)
        .Field(nameof(PhoneNumber))
        .Field(nameof(ReservationDate))
        .Field(new
FieldReflector<ReservationDialog>(nameof(ReservationDialog.ReservationT
ime)))
```

### Detailed Steps

```
.SetPrompt(PerLinePromptAttribute("What time would you like
to arrive?"))
).AddRemainingFields()
.Build();
}
```

We use the **IForm** of type **ReservationDialog** 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<ReservationDialog> BuildForm()
{
    return new FormBuilder<ReservationDialog>()
        .Field(nameof(Name))
        .Field(nameof(Email), validate: ValidateContactInformation)
        .Field(nameof(PhoneNumber))
        .Field(nameof(ReservationDate))
        .Field(new FieldReflector<ReservationDialog>(nameof(ReservationDialog.ReservationTime))
            .SetPrompt(PerLinePromptAttribute("What time would you like to arrive?"))
            .AddRemainingFields()
            .Build());
}
```

29. 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.

----- SNIP12-----

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

### Detailed Steps

```

        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
        };
    }

```

30. To wire this up, we need to go back to our RootDialog.cs. Add the following code to the StartAsync method. (Replacing what is currently there)

----- SNIP13-----

```

await context.PostAsync("Welcome to Dinner Bot, lets book a table for you. You will need to provide a few details.");
var reservationForm = new FormDialog<ReservationDialog>(new ReservationDialog(),
ReservationDialog.BuildForm,
FormOptions.PromptFieldsWithValues);
context.Call(reservationForm, ReservationFormComplete);

```

After a message to the user, we create a new form (reservationForm) of type ReservationDialog. Calling BuildForm. We then use context.call passing in both the form to be filled out, and a callback method.

In the FormOptions we use PromptFieldsWithValues otherwise it will use our initial hello as the first value.

```

await context.PostAsync("Welcome to Dinner Bot, lets book a table for you. You will need to provide a few details.");
var reservationForm = new FormDialog<ReservationDialog>(new ReservationDialog(),
ReservationDialog.BuildForm,
FormOptions.PromptFieldsWithValues);
context.Call(reservationForm, ReservationFormComplete);

```

We are almost there, we need to create the callback method. This is where we can see the results generated by the FormFlow.

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

### Detailed Steps

----- SNIP14-----

```
private async Task ReservationFormComplete(IDialogContext context,
IAwaitable<ReservationDialog> 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 ==
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;
        }
        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());
    }
}
```

### Detailed Steps

```

        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.

```

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.



## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

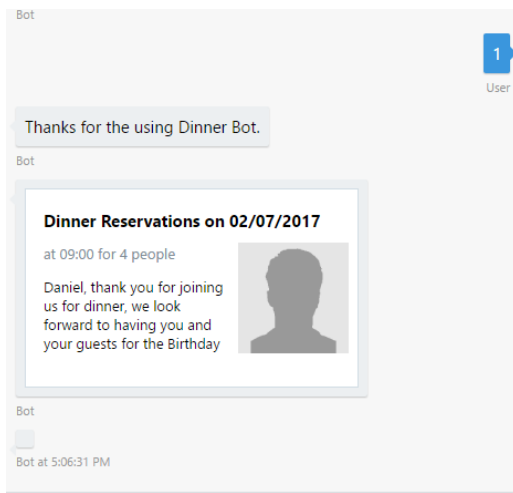
Page 33 of 42

### Detailed Steps

```
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.



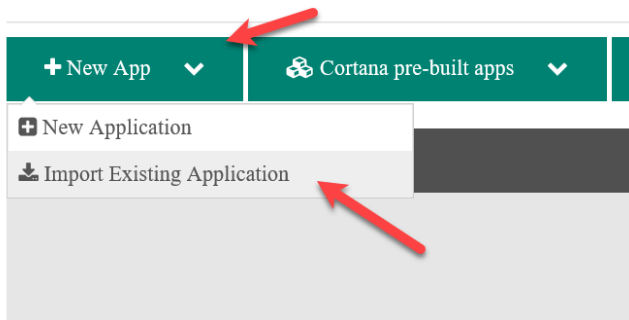
In the next exercise, we are going to tie all of this up to LUIS to get Natural Language Processing as part of your bot.

### Exercise 4: Using Intent Dialogs (LUIS)

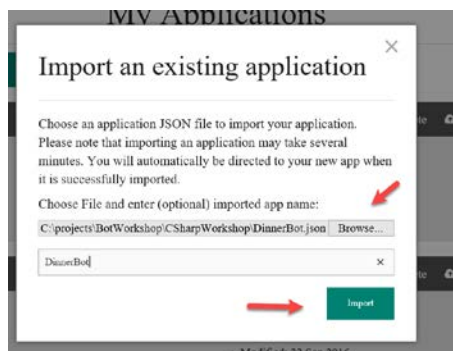
In this exercise we will import a LUIS Model that will handle questions coming from the users and route them to the appropriate Dialogs. We will not be creating the model but importing an already existing model. If you would like to learn how to create your own model you can find great tutorials and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

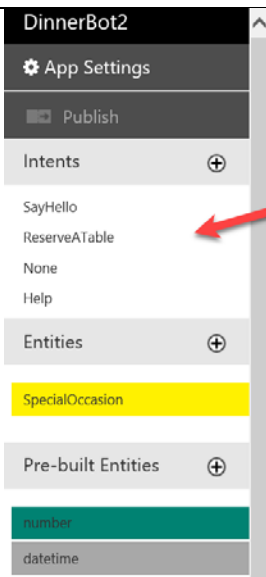


Once Imported you will notice that it has Intents for SayHello, ReserveATable, None, and Help and an entity of SpecialOccasion.

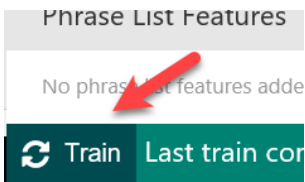
## Creating a bot using the Microsoft Bot Framework

Node.js Hands-on Labs

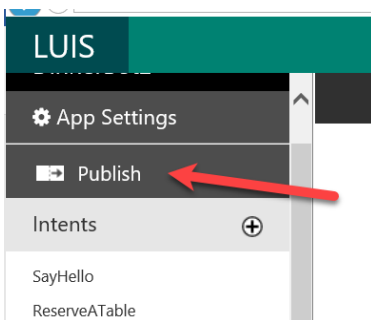
Page 35 of 42



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)

## Creating a bot using the Microsoft Bot Framework

C# Hands-on Labs

Page 36 of 42

URL for access via HTTP



Publish web service

able action fulfillment in one of your intents.

oft Bot Framework

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

ccess via HTTP



Update published application

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.

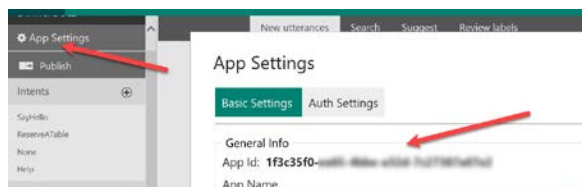
```
using Microsoft.Bot.Builder.Luis;  
using Microsoft.Bot.Builder.Luis.Models;
```

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

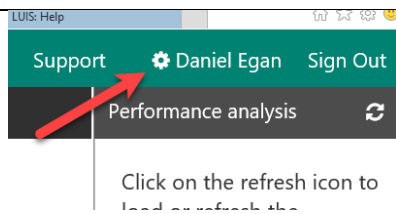


9. To get your Subscription key, click on the gear icon next to your name on the LUIS.ai website.

## Creating a bot using the Microsoft Bot Framework

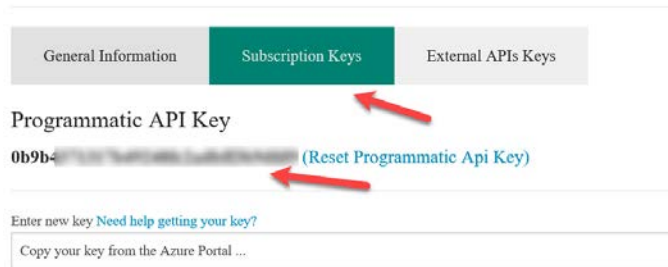
Node.js Hands-on Labs

Page 37 of 42



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

### My Settings



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)

```
namespace DinnerBot.Dialogs

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

[Serializable]
[LuisModel("d15aee24-", "0b9b-")]
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>
{

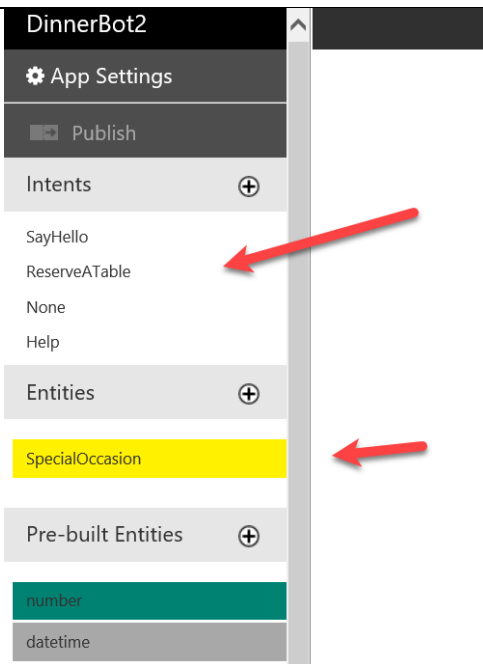
[LuisModel("d15aee24-", "0b9b-")]
public class RootDialog : LuisDialog<object>
{
```

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

## Creating a bot using the Microsoft Bot Framework

C# Hands-on Labs

Page 38 of 42



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<>

-----SNIP5-----

```
[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.

## Creating a bot using the Microsoft Bot Framework

### C# Hands-on Labs

Page 40 of 42

```
[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, FormOption);
        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

---

Information in this document, including URL and other Internet Web site references, is subject to change without notice and is provided for informational purposes only. The entire risk of the use or results from the use of this document remains with the user, and Microsoft Corporation makes no warranties, either express or implied. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 2009 Microsoft Corporation. All rights reserved.

Microsoft and Windows are trademarks of the Microsoft group of companies.

All other trademarks are property of their respective owners.