# 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](#_Toc482962274)

[Hands-on Lab Manual 1](#_Toc482962275)

[Lab Introduction 3](#_Toc482962276)

[Objectives 3](#_Toc482962277)

[Prerequisites 3](#_Toc482962278)

[Lab Scenarios 3](#_Toc482962279)

[Configuration and Setup 4](#_Toc482962280)

[Copy/Paste of Code 9](#_Toc482962281)

[Exercise 1: Basic Bot using BotBuilder 10](#_Toc482962282)

[Exercise 2: Creating Dialogs 16](#_Toc482962283)

[Exercise 3: Form Flow 29](#_Toc482962284)

[Exercise 4: Carousels, Cards, and Adaptive cards 44](#_Toc482962285)

[Exercise 5: Using Intent Dialogs (LUIS) 58](#_Toc482962286)

[Additional Resources 67](#_Toc482962287)

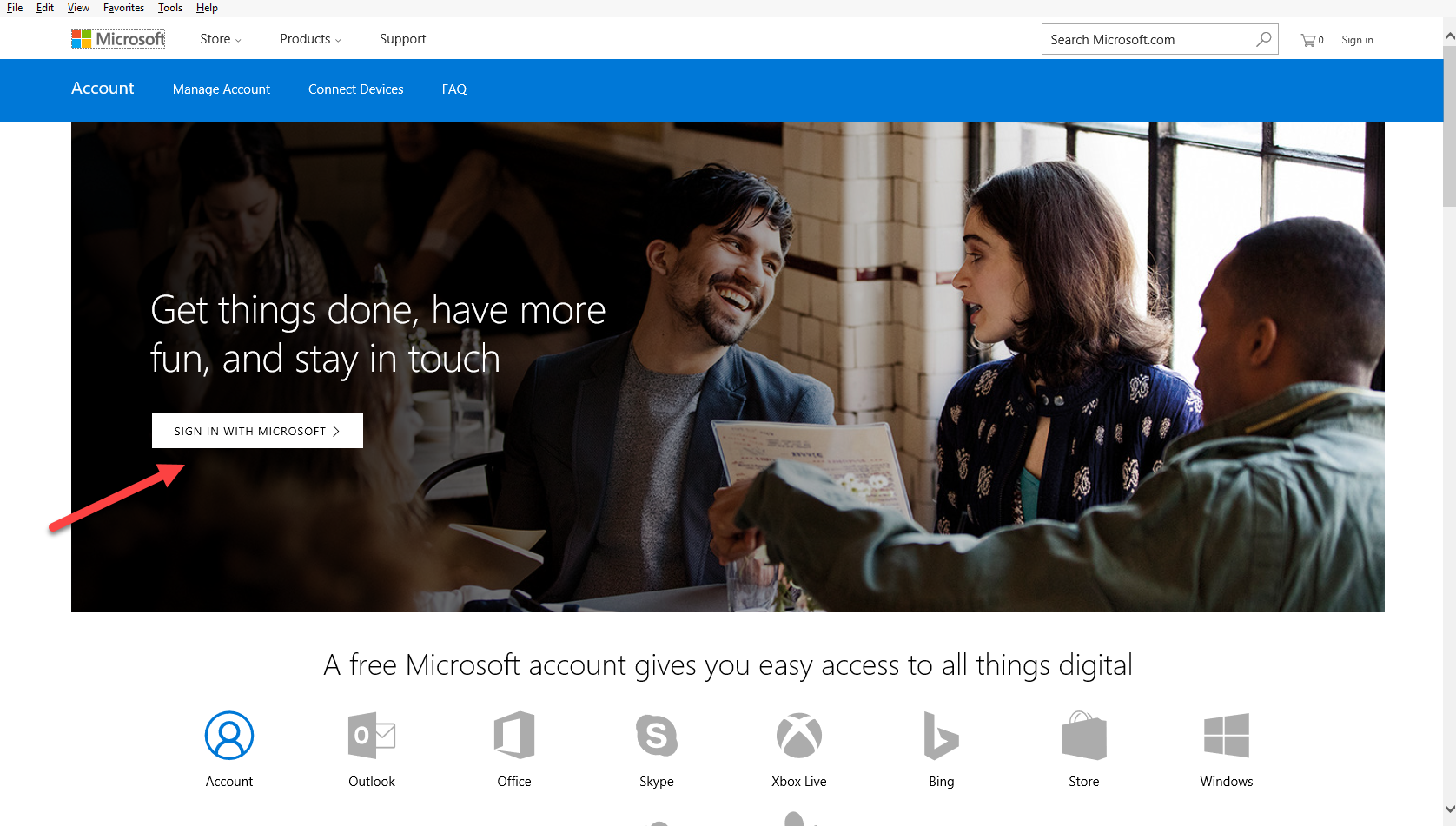
[Copyright 68](#_Toc482962288)

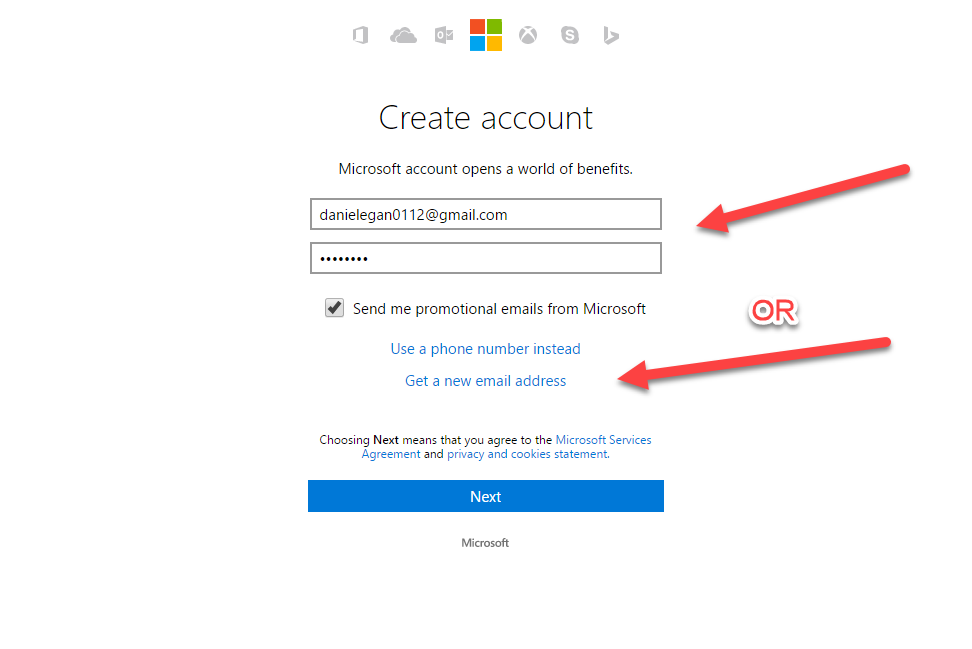
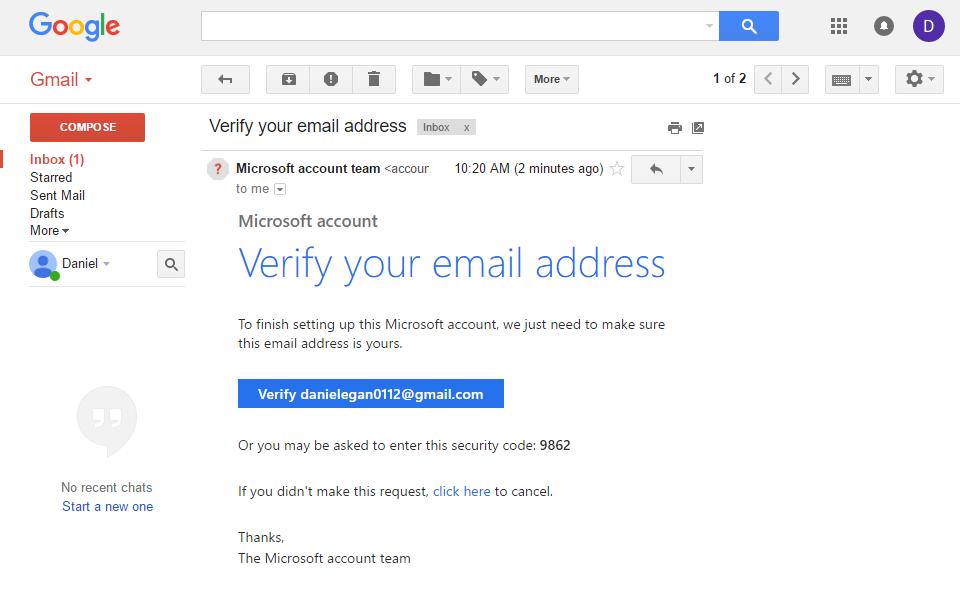
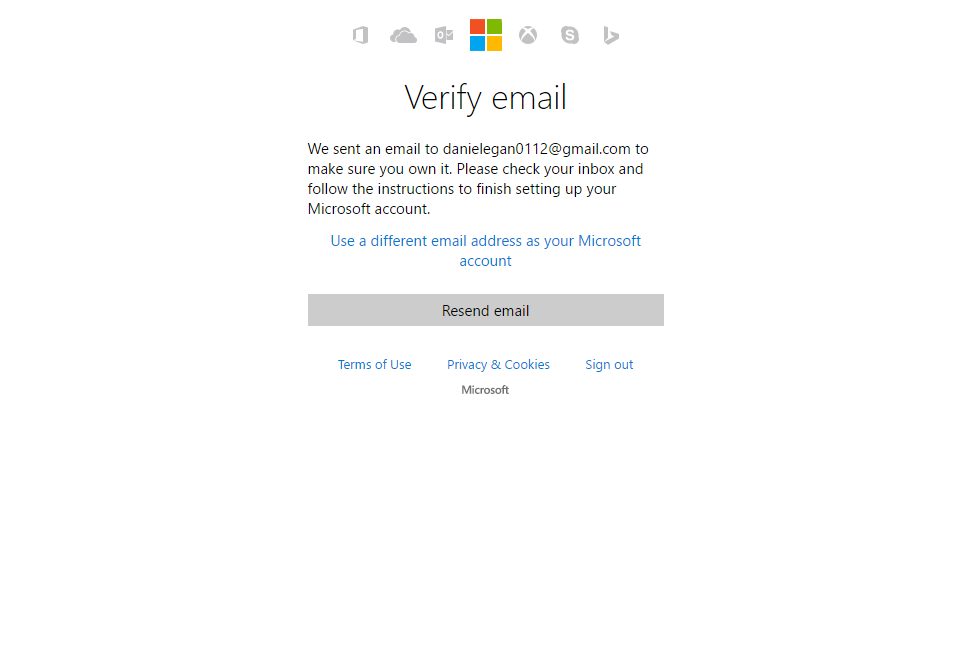
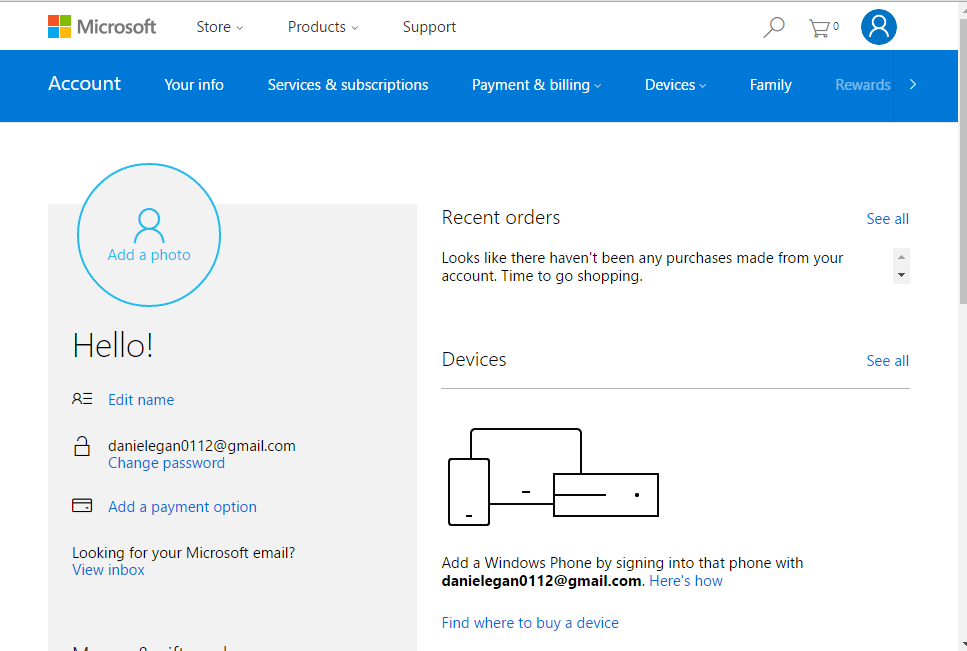
## Lab Introduction

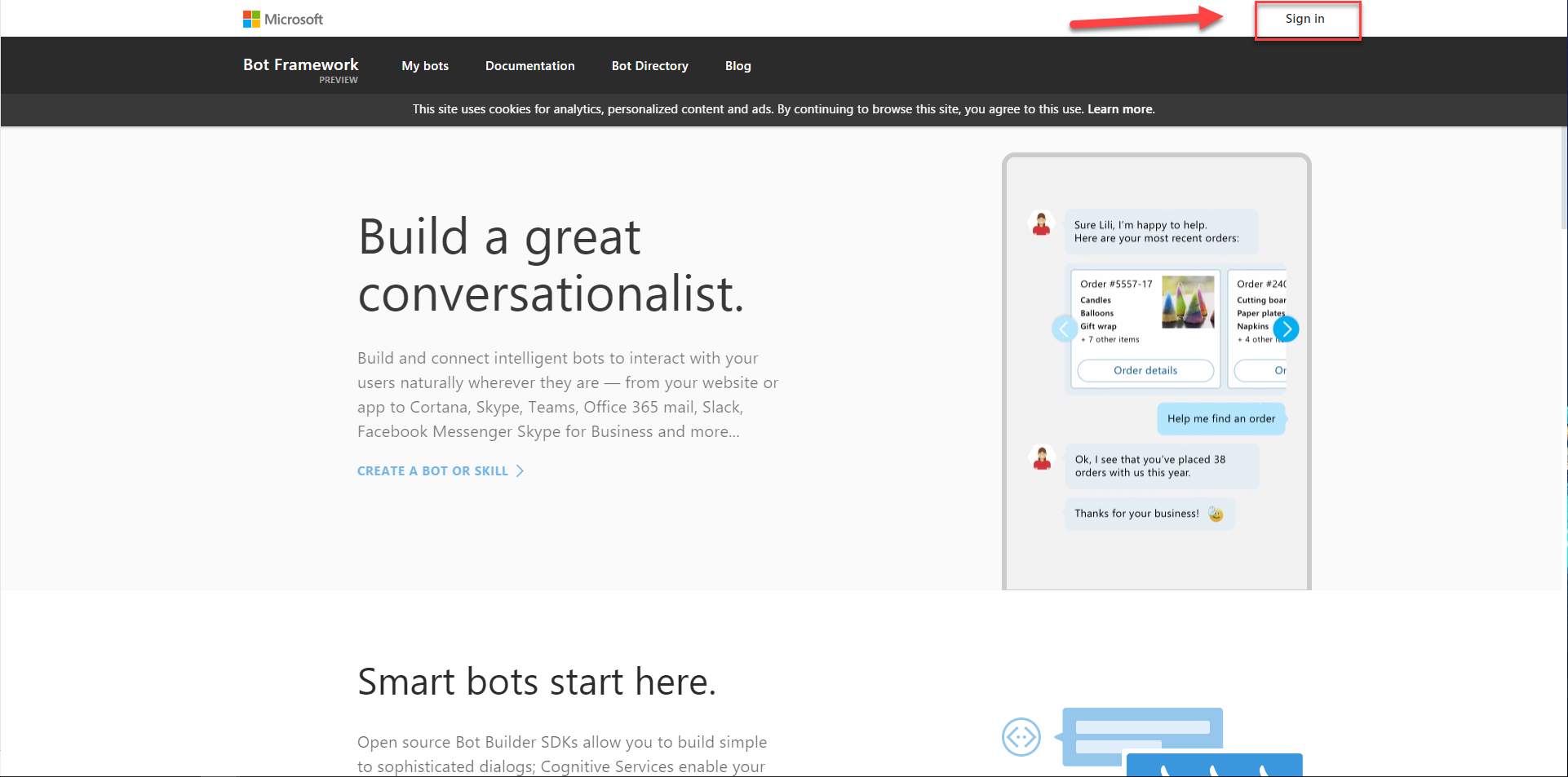
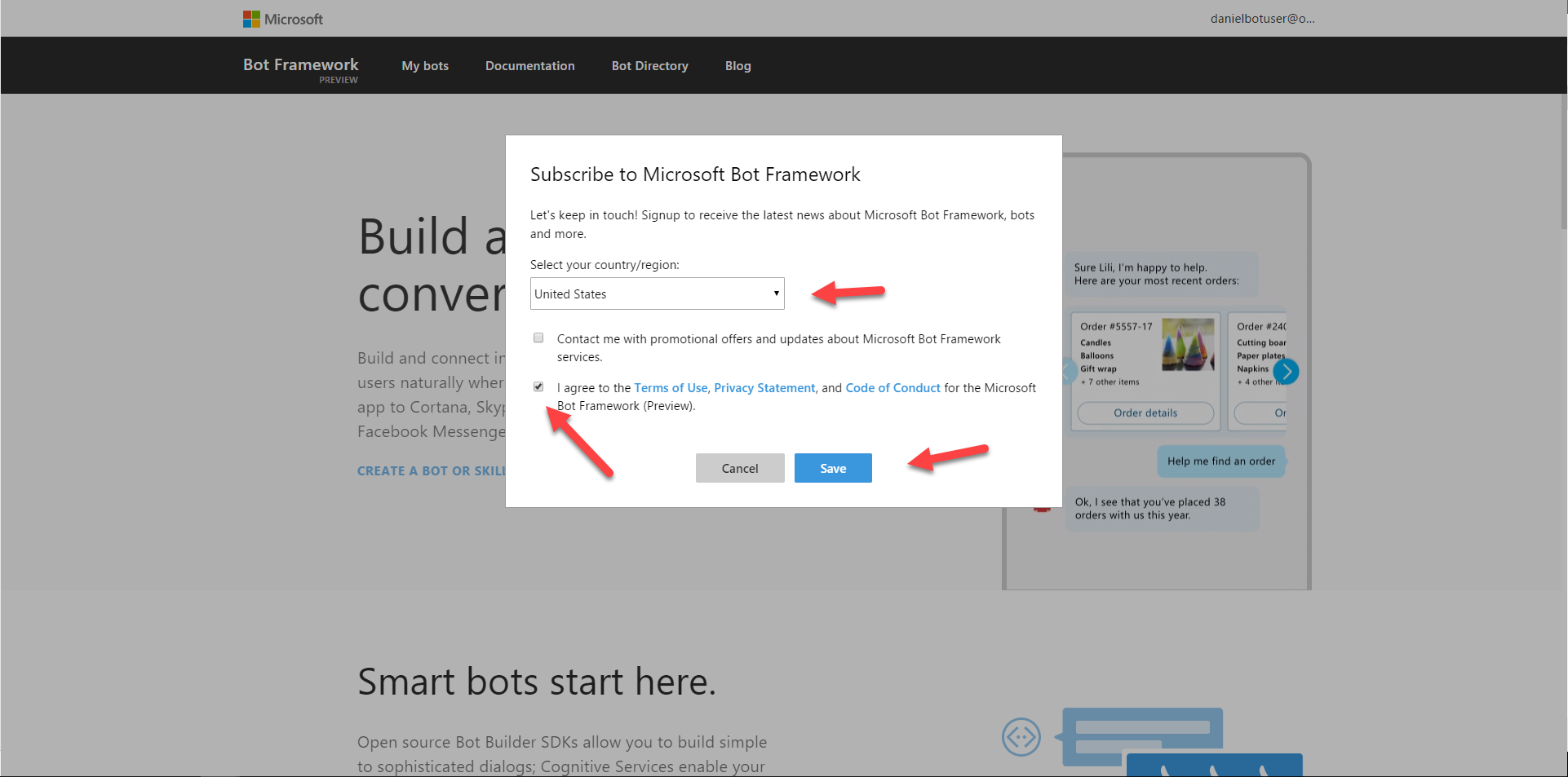
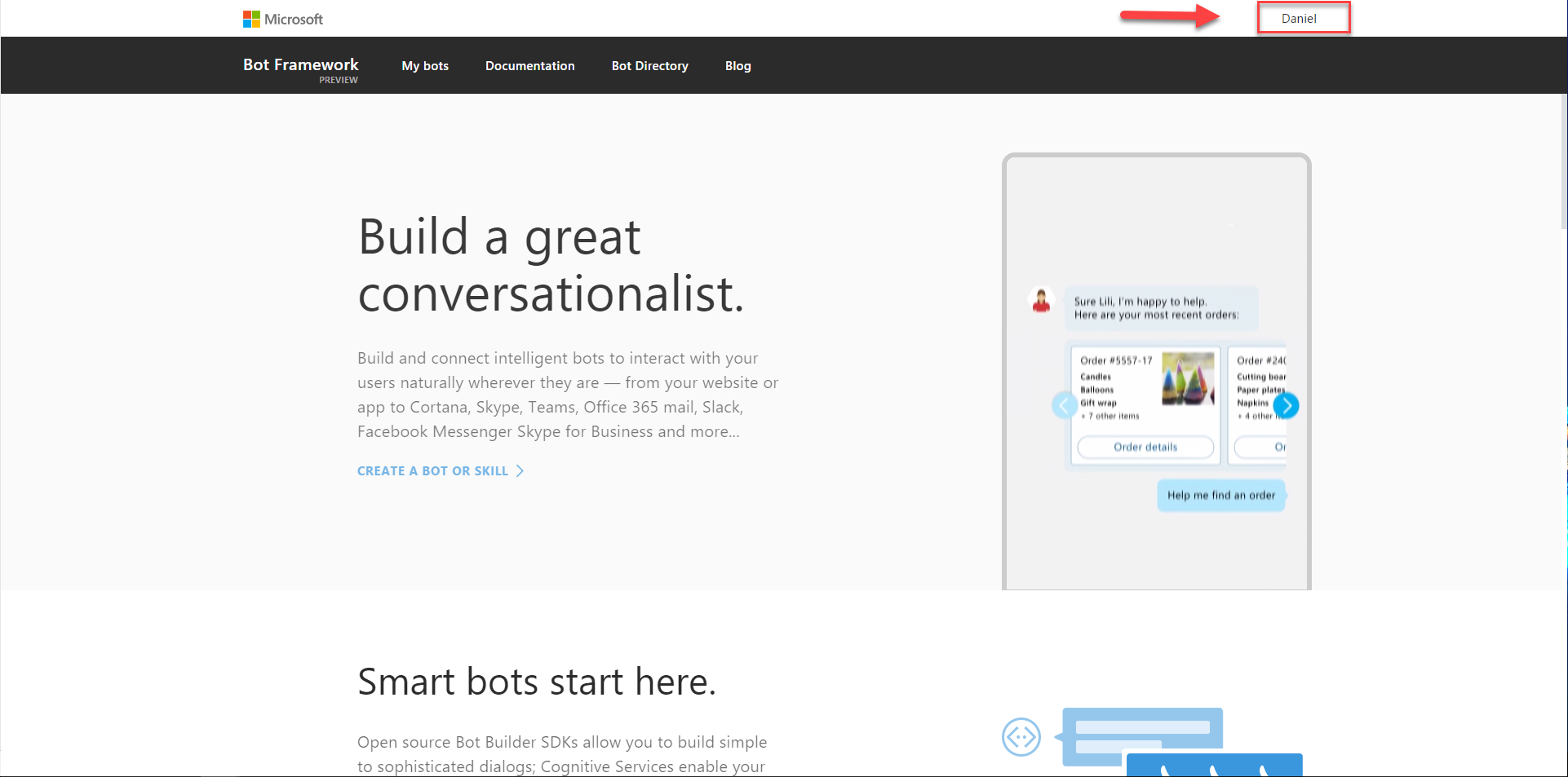
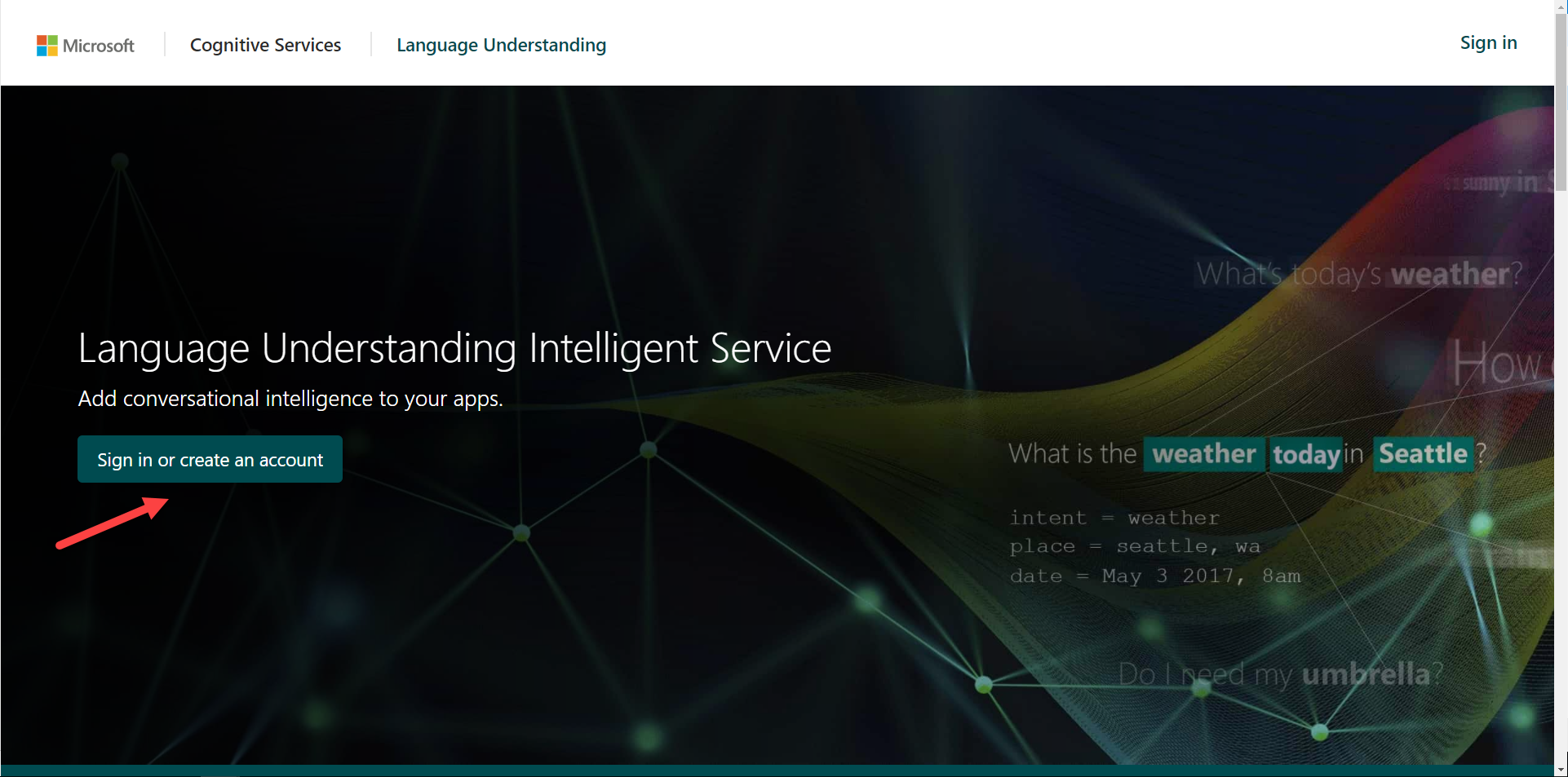
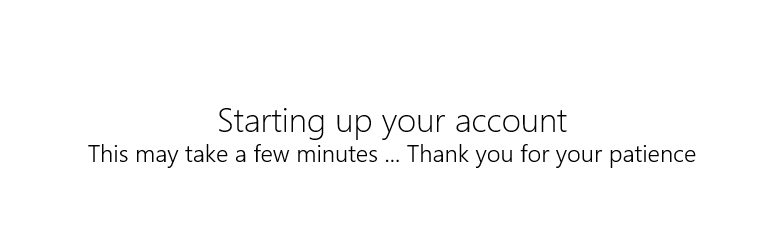
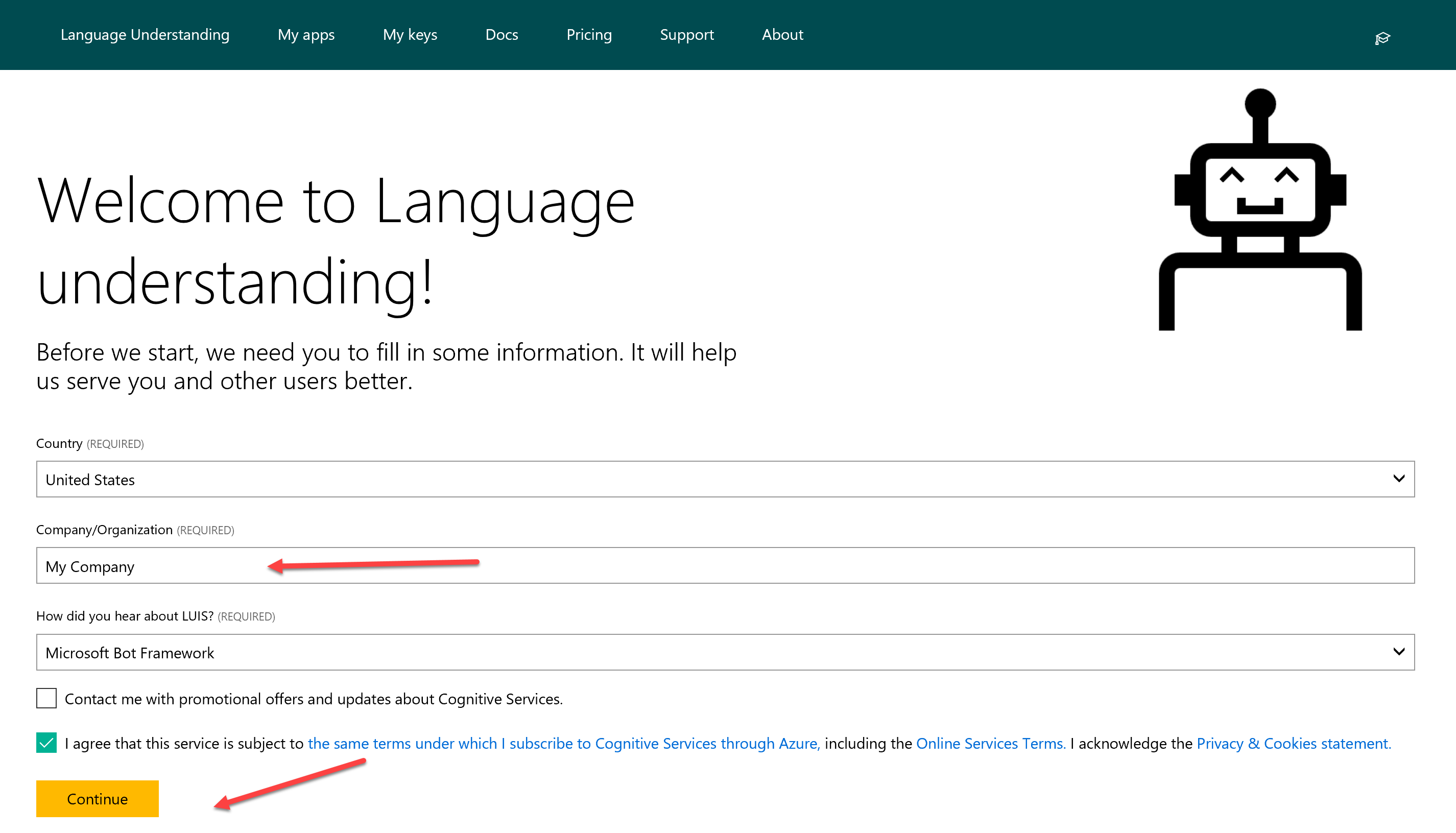
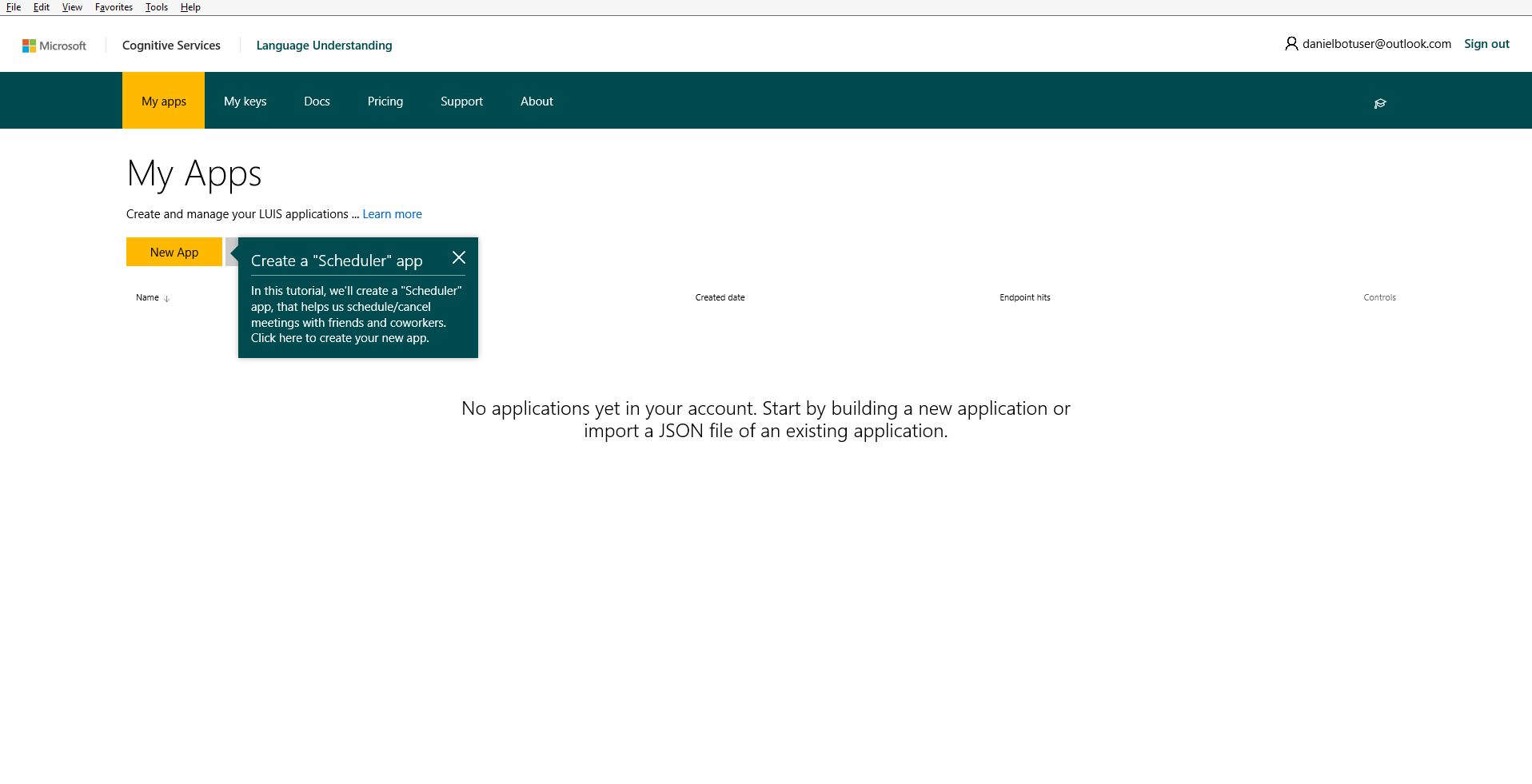
|  |  |
| --- | --- |
| Objectives | After completing these self-paced labs, you will be able to:   * Have an understanding of the basics of the Bot Framework |
| Prerequisites | * Visual Studio 2015 (community edition or other editions) * NGrok * Bot Application Template * Basic understanding of C# |
| 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 2017** : <https://www.visualstudio.com/vs/community/>
   * **C# Bot Application Template**: <http://aka.ms/bf-bc-vstemplate> When this zip is downloaded, copy (not unzipped) to %USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C#
   * **Update all Visual Studio Extensions (Tools 🡪 Extensions and Updates 🡪Updates)**
   * **Bot Framework Emulator:**  <https://docs.microsoft.com/en-us/bot-framework/resources-tools-downloads>
   * **Create a Microsoft ID** (if you don’t already have one)

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


* + Click on the **Create One** link.  
      
    
  + In the User name box enter your existing email address, or click Get a new email address to create an Outlook or Hotmail address.   
      
      
    **NOTE**: **If you use an existing email address you will need to verify it before moving on.**
  + Either path will take you to this screen  
      
    

1. Create a BotFramework account
   * Navigate to <http://BotFramework.com>
   * Click on sign in  
       
     
   * If you are using the same browser that you used to create your Microsoft ID then you will be signed in automatically, otherwise you will need to use the ID you just created to sign in.
   * Check the Terms of use box and click on Save.  
       
       
       
     
   * You can leave this window open, we will be using it later.
2. Sign-up for LUIS. Language Understanding Intelligent Services
   * <https://www.luis.ai/>
   * Click on: Sign in or Create Account button  
       
     
   * Sign in with your Microsoft account   
       
     
   * If you are still signed in it will ask you to say Yes to accept permissions. Otherwise you will need to sign in with the Microsoft ID you created earlier.
   * Fill out the required information (Put anything for company) and click Continue. (After it spins up)  
       
       
       
       
       
     That is all we need for now. We will come back to LUIS in another lab.   
     
   * We will explain and use this later for our bot.

### Copy/Paste of Code

You will have the option to copy/paste code snippets from this document to complete this lab.  You will learn much more by typing it in yourself but sometimes in a lab format speed is needed to get through all the exercises in time.   
  
**NOTE**: If you are on a mac, you will be using the PDF file. Do not copy and paste from the PDF file. There is a separate file called SNIPSCSharp.txt that contain the snips you need.

## Exercise 1: Basic Bot using BotBuilder

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

| **Detailed Steps** |
| --- |
| If you have not already done this in the prerequisites section, you will need to download and install the C# Bot Template. <http://aka.ms/bf-bc-vstemplate> (see instructions in Configuration and Setup section above)   1. Open or restart Visual Studio 2015 and go to **File** 🡪 **New** 🡪 **Project** Select the Bot Application Template and Name it DinnerBot    If you have used Web API previously, you will notice that the project that was set up is very similar to a WebApi project.   You can see both a **MessagesController** (which we will look at in a second) and a **WebApiConfig** in addition to a **RootDialog**.. Let’s open up the **WebApiConfig.cs**    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**  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.   C:\Users\danie\AppData\Local\Temp\Image.png  **NOTE**: If your **MesasgeController.cs** file does not look like this and you don’t have a RootDialog.cs file in the dialogs folder, you are using the OLD template. Delete it from %USERPROFILE%\Documents\Visual Studio 2015\Templates\ProjectTemplates\Visual C# and follow the instructions in the Configuration and Setup section above.  In this template, the main activity, message is handled here in the post. While all others are handled in the HandleSystemMessage below.  The **Post** message is marked with **asyc** because Bot Builder uses the C# facilities for handling asynchronous communication. So once we know it’s a **Message**, we call **Conversation.SendAsync** and send the activity to a new **RootDialog**. The **RootDialog** will be the first stop for everything we will be doing in this bot.   We will be making changes to this bot but first we need to make sure that we can test it using the emulator. Make sure you have downloaded (<https://docs.microsoft.com/en-us/bot-framework/resources-tools-downloads> ) and installed it before you begin.   1. In Visual Studio, place a couple of breakpoints in the **MessagesController.cs** file so we can inspect things when we connect. 2. 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. 3. Run the Bot Framework Channel Emulator that you previously installed.    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. 4. Click on the “Enter your endpoint URL” section to connect to your bot. 5. 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. 6. Click on **CONNECT**. If all goes well, you should see 200 [ConversationUdate] twice in your log. Once for the user and once for the bot. 7. Next, type Hello (or anything you want) into the txt field of the emulator.   Once you hit enter, you should hit the breakpoint you set in Visual Studio.     we are not going to walk through it, but take time to inspect the different values, properties and methods of the **Connector**, **Activity**, and **Message**.   When you are done, remove the breakponts and it **F5** to continure.   If you return back to the emulator, you will see the reponse from the bot (1), the entries in the log (2) and if you click on any of the post links, you will see the details associated with the request (3)    So in this section, we created a default hello world type of bot, got it up and running and interacted with it using the emulator. In the next section, we will start modifying it to create our dinner bot. |

## Exercise 2: Creating Dialogs

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

| **Detailed Steps** |
| --- |
| The first dialog has already been created for you from the template. This is the RootDialog (found in the Dialogs Folder). This will be the place where all of our interaction flows.  Let’s take a look at this file.   1. Double Click on RootDialog.cs to bring it up.    First notice that we mark the class as **[Serializable]**. The dialog stack and the state of all active dialogs are serialized to the per-user, per-conversation **IBotDataBag**. The serialized blob is persisted in the messages that the bot sends to and receives from the Connector. To be serialized, a Dialog class must include the [Serializable] attribute. All **IDialog** implementations in the Builder library are marked as serializable.     Next we implement the **IDialog<>** Interface. This interface has only one method **StartAsync** which is called when we create an instance of this dialog.     The **StartAsync** method calls **IDialogContext.Wait** with the continuation delegate to specify the method that should be called when a new message is received (**MessageReceivedAsync**). It is important to understand that the bot will wait here until the user sends a message. Then it will go to MessageReceivedAsync.    In our sample we are simply just echoing back what the user said to the bot with the length of characters sent. We will be changing this. Keep in mind that the RootDialog.cs should function like more of a traffic cop, directing to the dialogs that will perform functions.   To do this, we will need to create another dialog. 2. Right click on the Dialogs Folder and select **Add** 🡪 **Class** and name it **HelloDialog.cs**.     Once this comes up, we need to add a few using statements for the Bot. 3. Add the following using statements to the top of the **HelloDialog.cs** file.   **----- SNIP1-----------------------------------**   using Microsoft.Bot.Builder.Dialogs;  using Microsoft.Bot.Connector;  Next, we need implement the **IDialog** Interface.   1. Add the **IDialog<object>** interface to the **HelloDialog** class and implement the interface.    This will create a method called **StartAsync** which is what is called when we call the dialog. 2. 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. 3. Replace the default **NotImplementedException** with the following.   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 exit back to the RootDialog.   Now we need to make sure that this dialog is called from the **RootDialog**. 4. Open up the RootDialog.cs file and replace the code in the MessageReceivedAsync method with the following code .    We are using the context object to make a call out to the **HelloDialog**. We pass it the object (in this case a new **HelloDialog()** and a callback method for it to return to, called **HelloDialogCallback**. Let’s implement that. 5. Hover over the **HelloDialogCallback** and select Generate method 6. Replace the throw new NotImplementedException() with the following code and add the asyc classifier to the method.     Since we are not passing anything back from the dialog at this point, all we want to do is have it wait for input and ready to go to the MessageReceivedAsync method. Of course, at this stage, all it will do is loop back to HelloDialog again since that is the only dialog we have.   Let’s test our new dialog. 7. Hit **F5** or press the green arrow  to run your project. Make sure the browser launches. (And remove the breakpoint in the MessageController if it is still there) 8. Open up the emulator and click on the top bar to revel the last connection we used and select connect.   Once the emulator launches, type in hello and the bot will now use our HelloDialog. No matter what you type it will go there and return to the root again.     Now that we have this working, lets make the HelloDialog actually do something other then sending a simple text message.  In the HelloDialog we are going to show how to save state to the state bag. 9. Inside you **HelloDialog.cs** file, place the following code inside the StartAsync method replacing what we have in there.     **----- 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.   1. Paste the following code **below** the **StartAsync** Method    **----- SNIP3-----------------------------------**   private static async Task Respond(IDialogContext context)  {  //Variable to hold user name  var userName = String.Empty;  //check to see if we already have username stored  context.UserData.TryGetValue<string>("Name", out userName);  //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));  }  }   1. 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.     **----- SNIP4-----------------------------------**   public async Task MessageReceivedAsync(IDialogContext context, IAwaitable<IMessageActivity> argument)  {  //variable to hold message coming in  var message = await argument;  //variable for userName  var userName = String.Empty;  //variable to hold whether or not we need to get name  var getName = false;  //see if name exists  context.UserData.TryGetValue<string>("Name", out userName);  //if GetName exists we assign it to the getName variable and replace false  context.UserData.TryGetValue<bool>("GetName", out getName);  //If we need to get name, we go in here.  if (getName)  {  //we get the username we stored above. and set getname to false  userName = message.Text;  context.UserData.SetValue<string>("Name", userName);  context.UserData.SetValue<bool>("GetName", false);  }  //we call respond again, this time it will print out the name and greeting  await Respond(context);  //call context.done to exit this dialog and go back to the root dialog  context.Done(message);  }    The code is well commented, take your time to see how things are used in the dialog.   Now we want to wire up the **RootDialog** a little better in order to send the user into the **HelloDialog** and receive back data**.**   1. Open up the **RootDialog.cs** file and add two strings to the top of the class to represent the choices.      **----- SNIP5-----------------------------------**   private const string ReservationOption = "Reserve Table";  private const string HelloOption = "Say Hello";  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.   1. Paste the following code inside the **MessageReceivedAsync** method in the **RootDialog.cs** file. (REPLACING WHAT IS IN THERE)  This will let them choose between reserving a table or just saying hello.    **----- 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.     1. Since we are using a list, add the System.Collections.Generic using statement to the top of the file. 2. In the **RootDialog.cs** file place the following code below the **MessageReceivedAsync** method.      **----- 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($"Ooops! Too many attempts :( You can start again!");  //This sets us in a waiting state, after running the prompt again.  context.Wait(this.MessageReceivedAsync);  }  }  There are a couple of important parts of this code. If they selected the HelloOption then they will be sent to the **HelloDialog** by using **context.call**.     when it finishes that dialog it will return to the **ResumeAfterOptionsDialog** method as show in the code above so we will need to implement that method.   1. Paste the following code below the **OnOptionSelected** method in the **RootDialog.cs** file. In this code we are retrieving the message back from the Dialog (but doing nothing with it), capturing any errors coming back, and setting it ready for the user to communicate again with the call to context.wait.     **----- SNIP8-----------------------------------**   private async Task ResumeAfterOptionDialog(IDialogContext context, IAwaitable<object> result)  {  try  {  var message = await result;  }  catch (Exception ex)  {  await context.PostAsync($"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) .  You will notice the second time that I say hello. It does not ask for my name, but pulls it out of UserData.  If you look at the code in the **HelloDialog** you can see the potential for unintended use, meaning we are not checking values, of confirming, or validating data. We could of course write all that by hand but we don’t need to. In the next exercise, we will use FormFlow to help us with this. |