# +Creating a bot using the Microsoft Bot Framework In Node.js

# Hands-on Lab Manual

Table of Contents

[+Creating a bot using the Microsoft Bot Framework In Node.js 1](#_Toc473535331)

[Hands-on Lab Manual 1](#_Toc473535332)

[Lab Introduction 3](#_Toc473535333)

[Objectives 3](#_Toc473535334)

[Prerequisites 3](#_Toc473535335)

[Lab Scenarios 3](#_Toc473535336)

[Configuration and Setup 4](#_Toc473535337)

[Copy/Paste of Code 9](#_Toc473535338)

[Exercise 1: Basic Bot using BotBuilder 10](#_Toc473535339)

[Exercise 2: Using prompts in a bot 16](#_Toc473535340)

[Exercise 3: Using Intent Dialogs (LUIS) 19](#_Toc473535341)

[Exercise 4: Connecting to Skype and Webchat 25](#_Toc473535342)

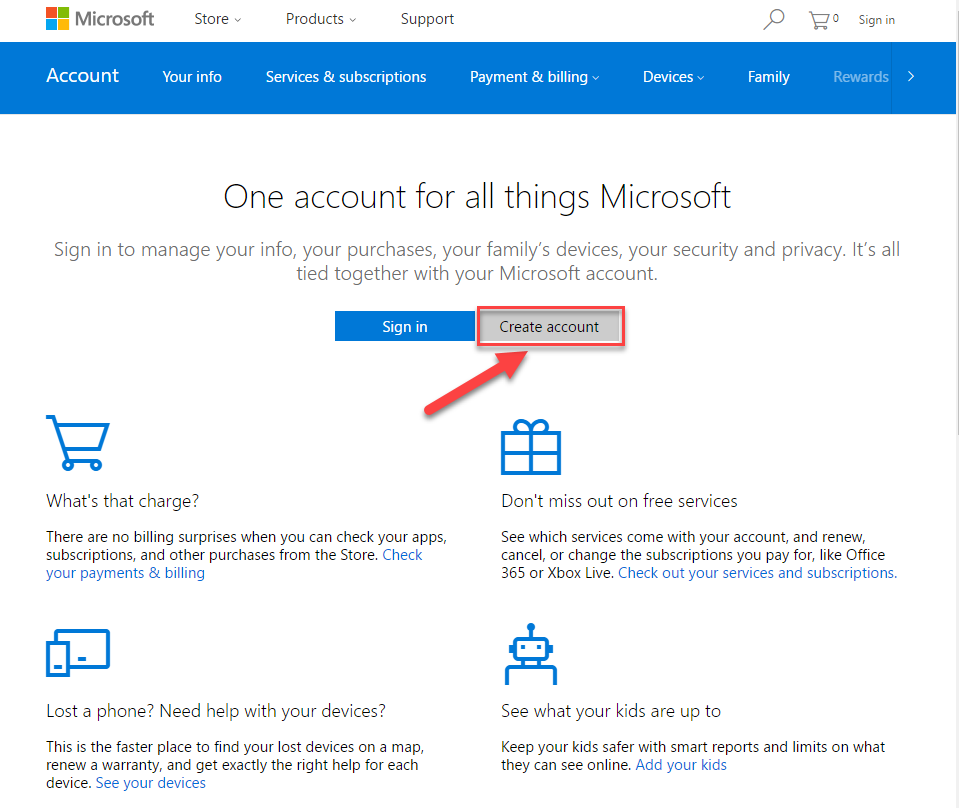
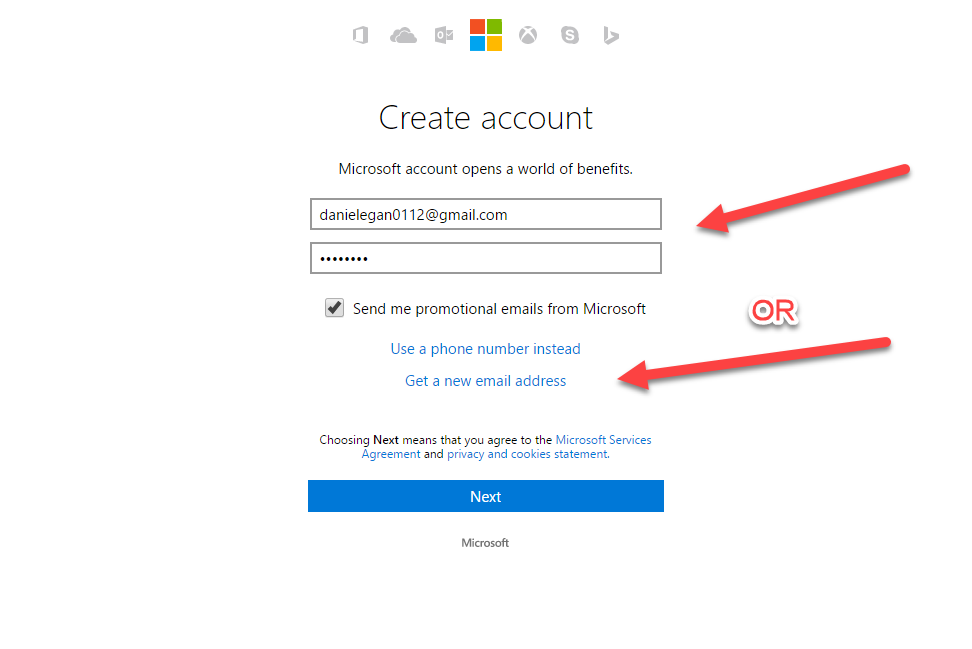
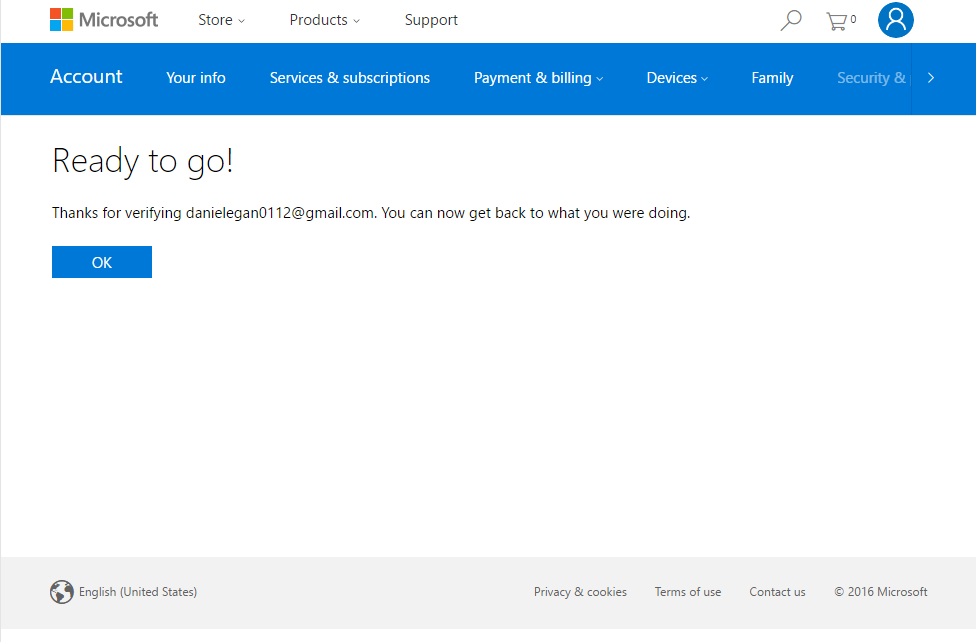
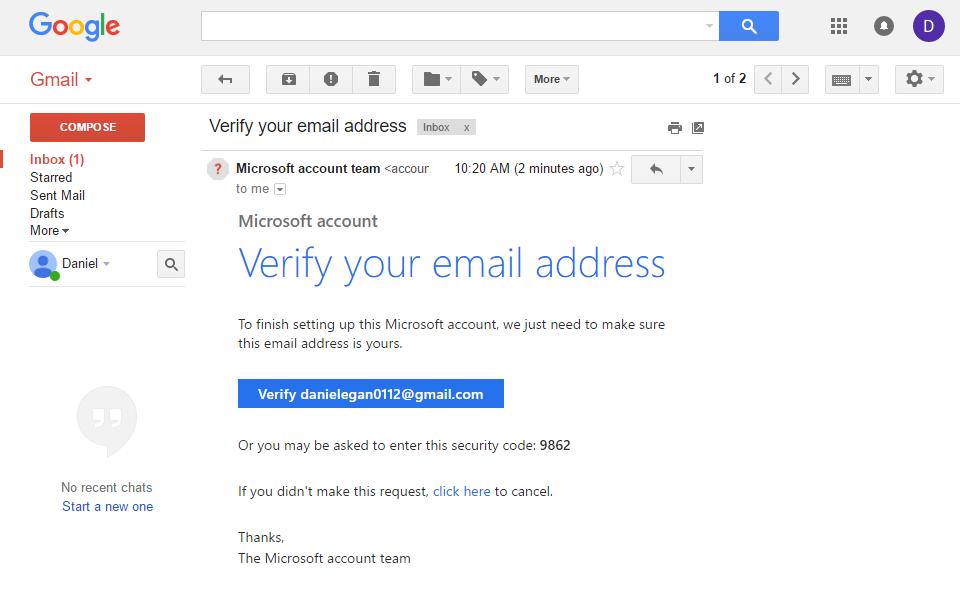
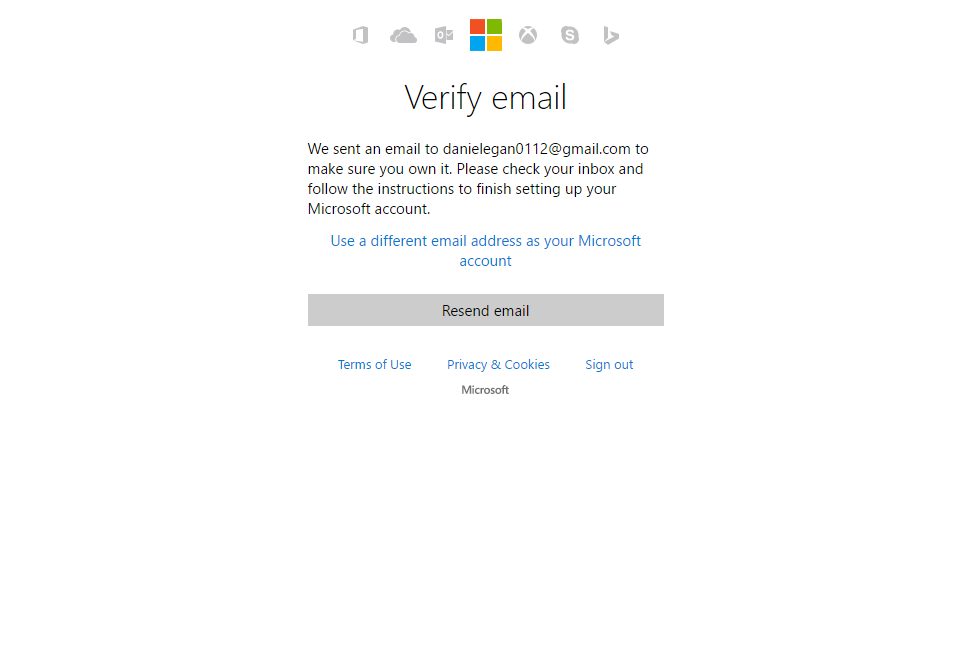
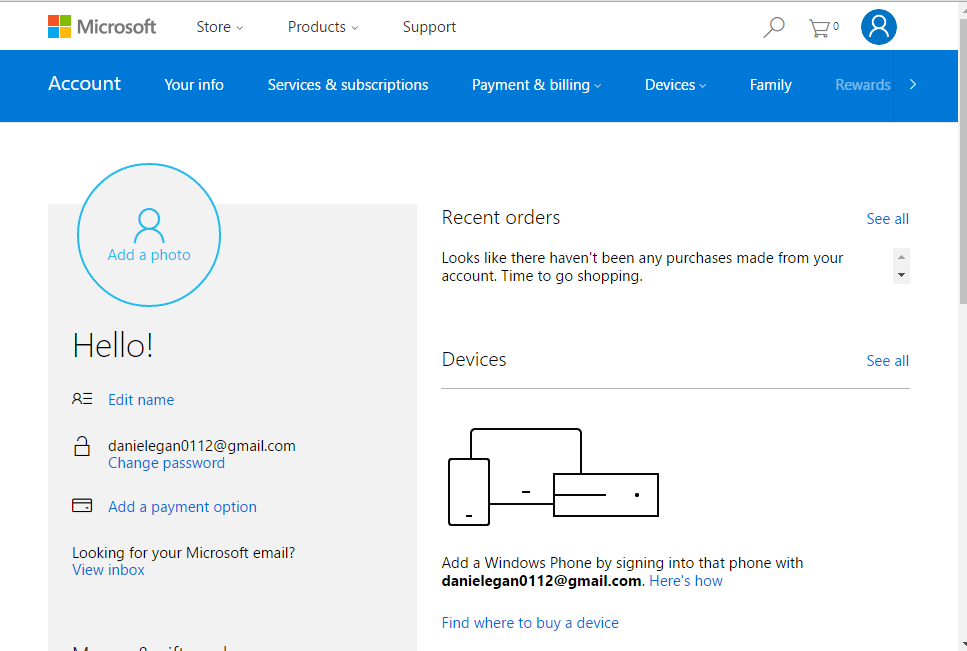
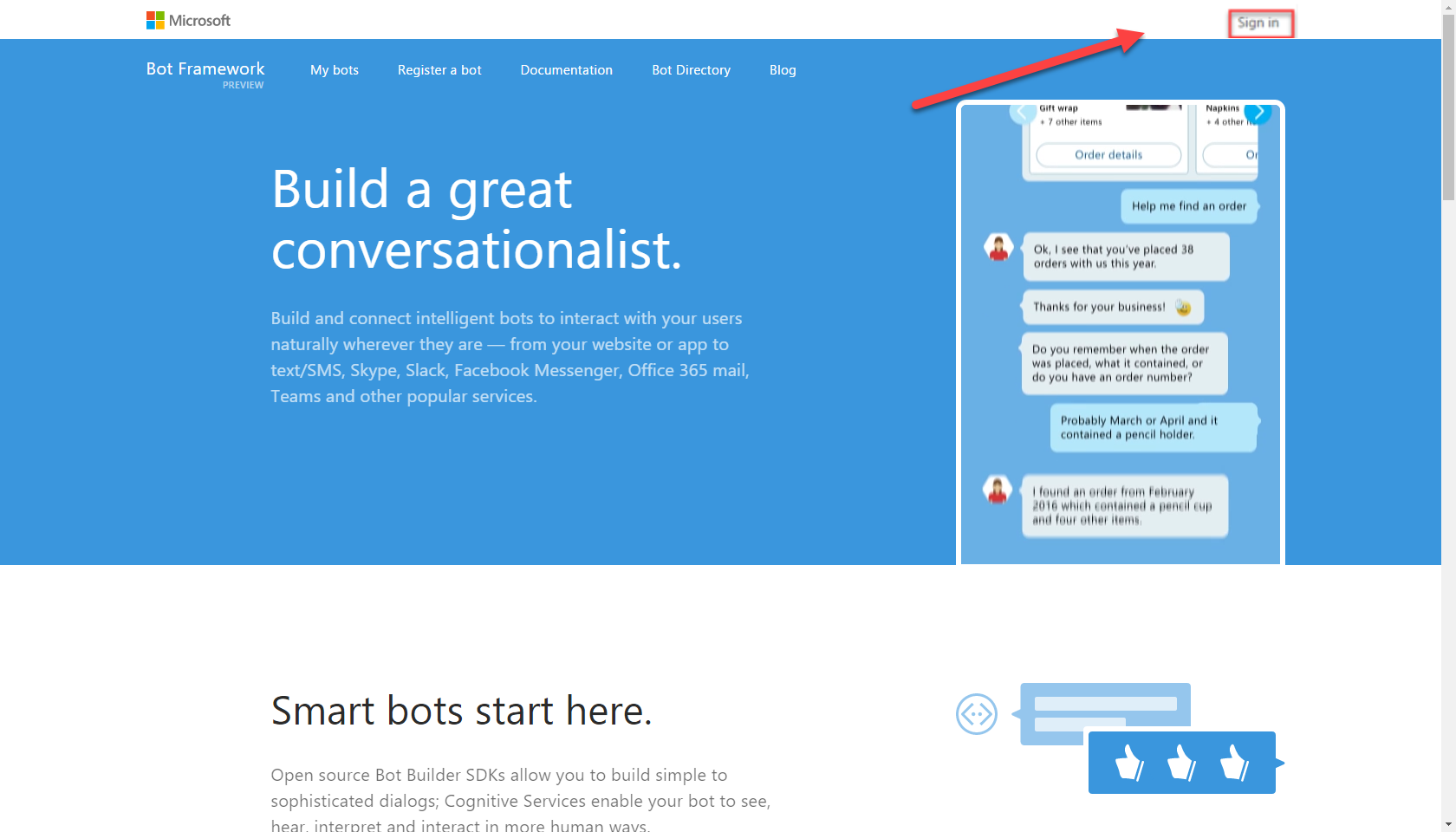
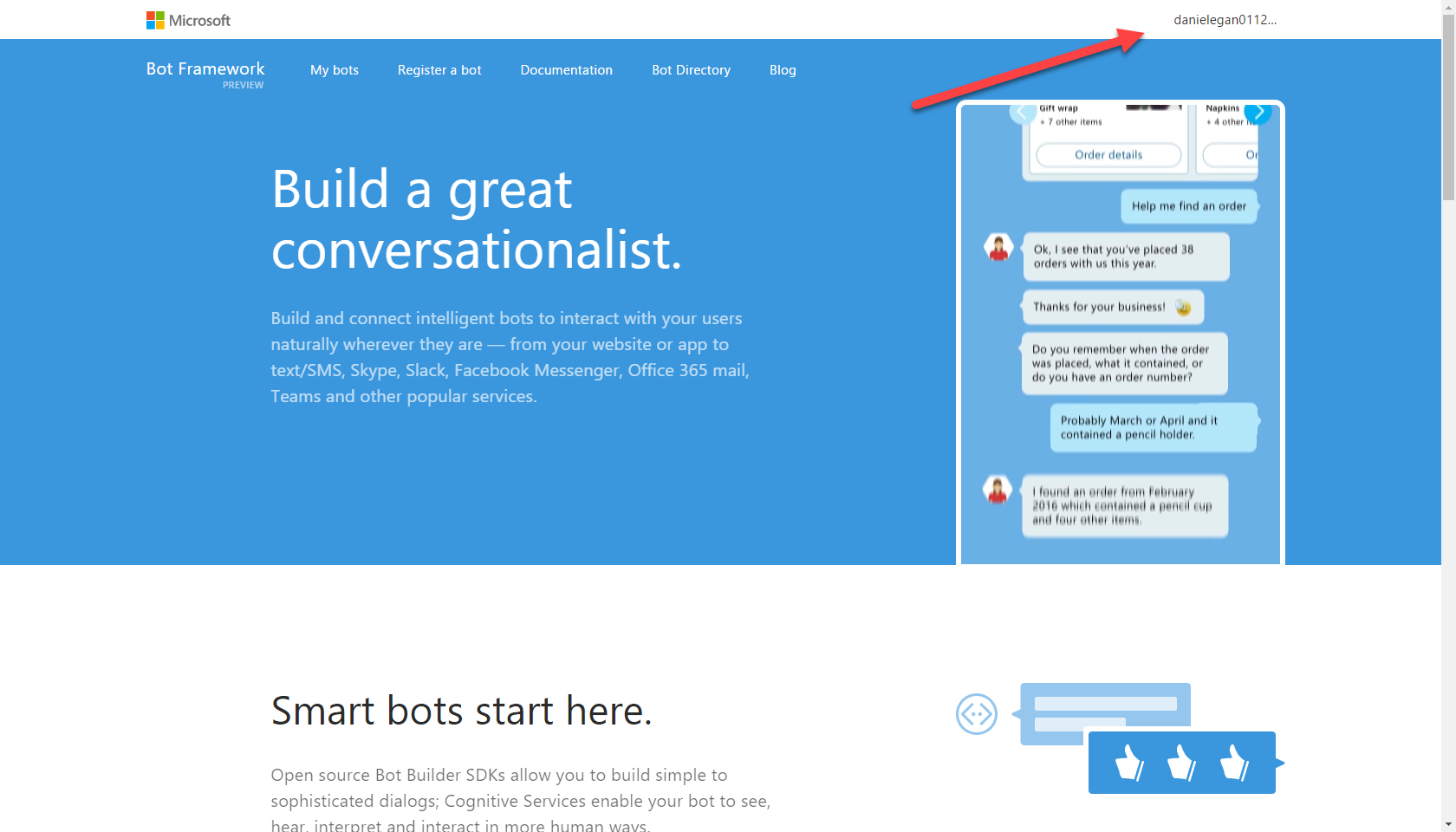
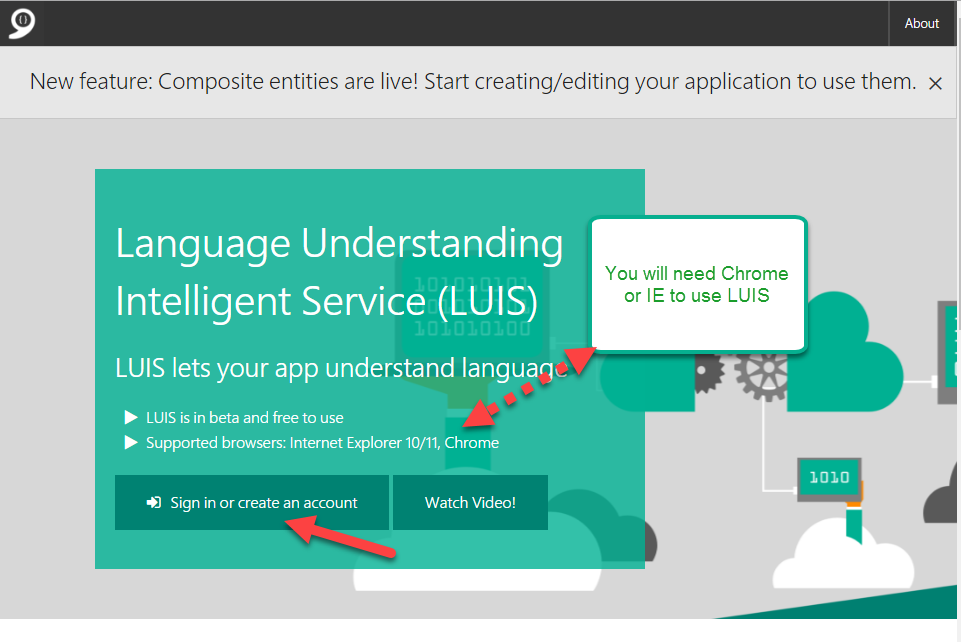
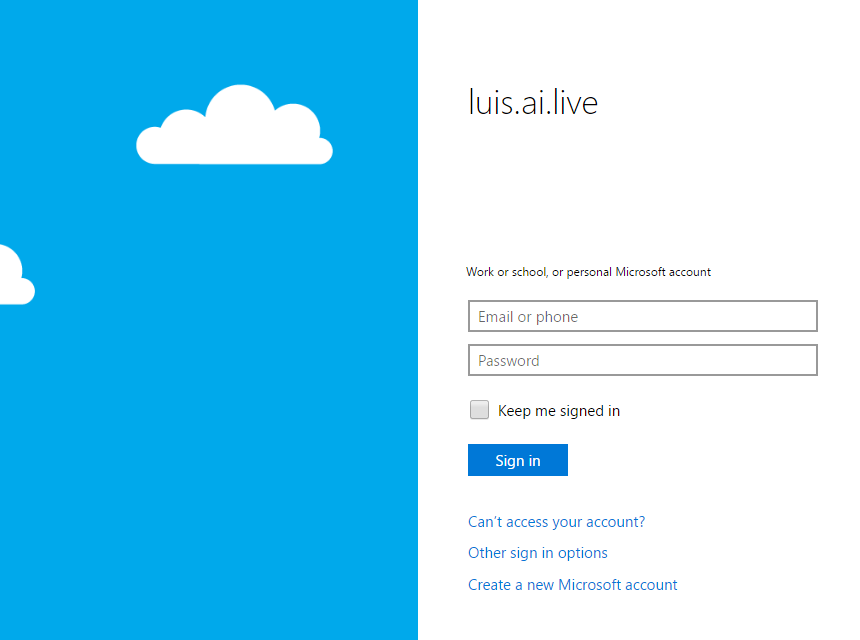
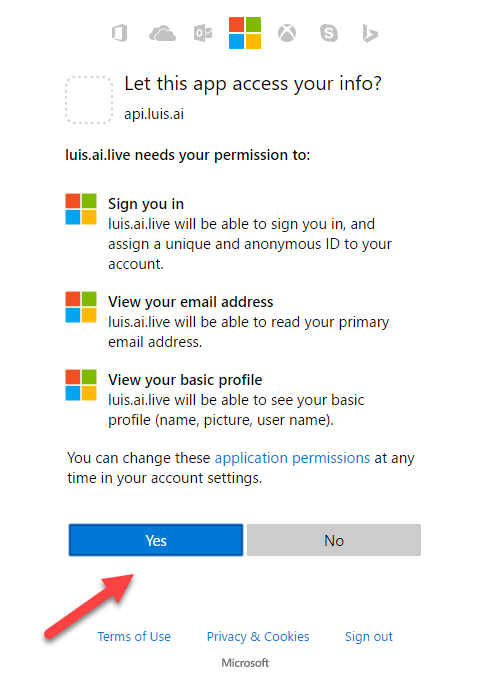
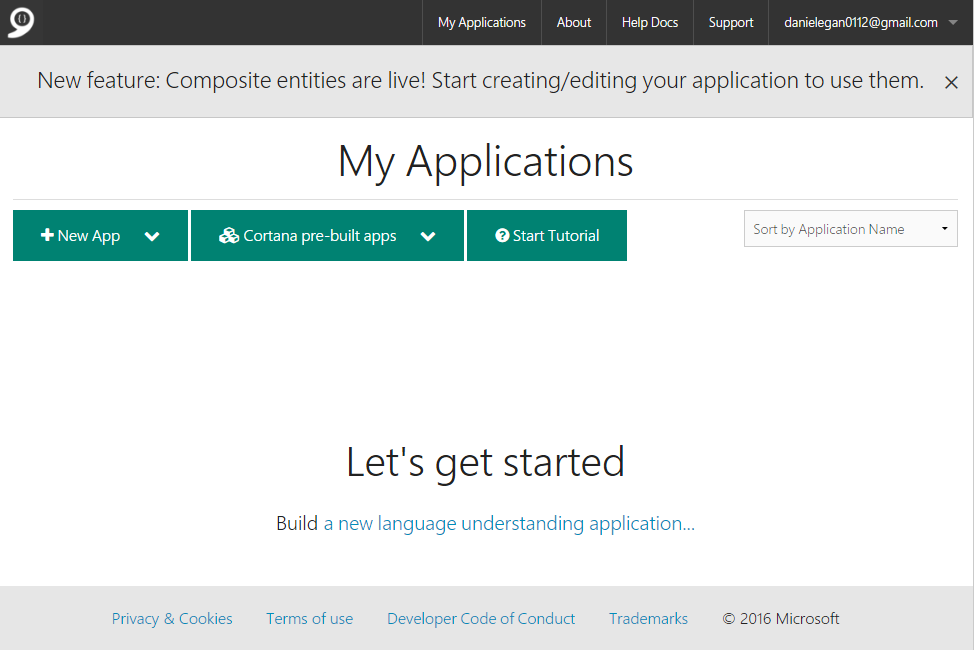
[Additional Resources 40](#_Toc473535343)

[Copyright 41](#_Toc473535344)

## 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 | * Node.js * A command line utility * Basic understanding of Node.js & JavaScript |
| Lab Scenarios | This series of exercises is designed to show you how to get started using the Microsoft Bot Framework. |

### Configuration and Setup

1. Install prerequisite software
   * **Node.js** : <https://nodejs.org/en/> (this will include NPM)
   * **NGrok** : <https://ngrok.com/>   
     **Skype** : <http://skype.com> (if you want to test a Skype Bot)
   * **Visual Studio Code**. VS Code is cross platform, you can use others if you like (like sublime, Atom, Notepad++, Vim, etc..) but you will be on your own with debugging and usage.
     1. <https://code.visualstudio.com/>
2. **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.   
       
       
     **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  
       
     
3. 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.   
       
     
   * You can leave this window open, we will be using it later.
4. 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.  
       
     
   * 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.  
       
     
   * 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 SNIPS.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** |
| --- |
| 1. Open up your Terminal. (This could be Terminal on Mac, PowerShell on windows, or a  terminal of your choice (cmndr, iTerm, etc…)  On Mac hit **Command 🡪 Spacebar** and type terminal  C:\Users\danie\AppData\Local\Temp\SNAGHTMLc8c305.PNG  From windows hit the windows key and type node or PowerShell 2. Next we want to create a folder to hold our project and add some initial files. (personally I hold all project folders in a folder called projects (**C:\Projects on PC or my ’userfolder’/projects** on mac).   To create the folder, type the following from the command line to make a directory (mkdir)  ~$ mkdir botworkshop  next we want to change to that directory (cd)  ~$ cd botworkshop  next we will initialize it with a **package.json** file. We will use the -**y** so that it gives us the default values.   ~$ npm init -y   the next step is to install the botbuilder npm module   ~$ npm install botbuilder --save   finally, we want to open all of this up in **VSCode** type the following (the word code with a space then a period)  ~$ code .   **NOTE**: You can also open up VSCode and select **File 🡪 Open (MAC) File 🡪Open folder (PC)** and select the botworkshop folder we created)  This will open up our project in VSCode (again, you are free to choose your own editor if you like) 3. Now we want to create our first simple hello world bot. Click on the **Add File** icon, and name the file **helloworld.js** 4. In the file you just created, either type in or paste the following code. **NOTE**: if you are using the PDF file, you can find the snippets in a separate file called SNIPS.txt   **------SNIP1-------------------------------------------------------------**   //Talking with the user.. as simple as possible  var builder = require('botbuilder');  var connector = new builder.ConsoleConnector().listen();  var bot = new builder.UniversalBot(connector);  bot.dialog('/', function (session) {  session.send('Hello World');  });   1. Now go back to your command prompt (terminal, powershell, etc) and type the following (make sure you are still in the projects/botworkshop folder)  ~$ node helloworld.js  Nothing should happen, Well, you should not get any errors anyway. If all went well, it should be “listening” for you.  Type the word Hello (or anything really) and you should get back a “Hello World” from your bot. 2. Hit **Ctl +C** to exit out of the bot and back to command line. 3. Now we want to setup a debug session. It is best to do this on the simple bot so you can get used to using the debugger.   **>** In VSCode, click on the bug icon (Left side, second from bottom) **>** On the top you can see that there is no config file. Click on the green arrow top left **>** This will produce a drop down, select Node.js 4. This will create a **launch.json** file. Open this file and change the **“program”** attribute from **index.js** to **helloworld.js**     **NOTE**: The last step is not used in this HOL because we will be using Attach as opposed to LAUNCH but it is good to be aware of that field for the future. 5. Next open the **helloworld.js** file and put a breakpoint next to the **session.send(‘Hello World’);** line by clicking next to the line number. 6. Now we want to run our program in debug mode. Open up your console (Terminal, Powershell, etc) and type the following:  ~$ node --debug-brk helloworld.js 7. That should run the node process in debug mode listening on port 5858 8. In VS Code, make sure you are still in the debug panel, in the debug dropdown make sure you select **attach** (to attach to the process we just started that is running on port 5858) and click on the green debug arrow to run it. 9. This will attach to the process and the debug will be stopped on the first line of the program (that’s what using **--debug-brk** does as opposed to just **--debug**) you will be able to view all the local variables on the left, step through the code (Function keys or buttons on top) and inspect variables by hovering over them. You can hit F5 to run the program (or the green arrow on the top bar.   Spend time debugging and looking around in this simple example so you can debug more complex ones later.    That is the end of exercise one. Now that we have everything set up and a simple bot running we will talk about how to handle greater complexity when working with your bot. |

## Exercise 2: Using prompts in a bot

In this exercise we will create a very simple bot so that you can get used to using different prompts in a waterfall, and collecting data.

| **Detailed Steps** |
| --- |
| 1. Open up Visual Studio Code in the same folder (project) we started earlier. Add a new file by clicking on the new file icon and name it **promptbot.js** 2. First, we need to add the require for botbuilder, and to new up our connector and bot just like we have done before.  Add the following code to the top of the **promptbot.js** file.   **----- SNIP2-----------------------------------**   var builder = require('botbuilder');  var connector = new builder.ConsoleConnector().listen();  var bot = new builder.UniversalBot(connector);   1. Next we need to add our dialog. It is one dialog, so let’s look at the whole thing at once. (It makes it easier to copy/paste as well) Paste the following code directly below our bot variable **(var bot = new …)** in **promptbot.js**   **------SNIP3---------------------------------------**   bot.dialog('/', [  function (session) {  builder.Prompts.text(session, "Hello... What's your name?");  },  function (session, results) {  session.userData.name = results.response;  builder.Prompts.number(session, "Hi " + results.response + ", How many years have you been coding?");  },  function (session, results) {  session.userData.coding = results.response;  builder.Prompts.choice(session, "What language do you code Node using?", ["JavaScript", "CoffeeScript", "TypeScript"]);  },  function (session, results) {  session.userData.language = results.response.entity;  session.send("Got it... " + session.userData.name +  " you've been programming for " + session.userData.coding +  " years and use " + session.userData.language + ".");  }  ]);  In this code we are creating a function array (between the []) so it will cascade from one question to the next. Within the waterfall we are using 3 different prompts (text, number, choice) and storing data for the user in **userData**. We then pull the data out for the final **session.send** message.   1. Now we can run the code. Go to your command prompt and type in the following.  ~$ node --debug promptbot.js 2. Next go to Visual Studio Code and place some breakpoints in your code so you can step through and inspect the objects and flow. 3. Step through the code as we had before to watch as things are running. |

## Exercise 3: Using Intent Dialogs (LUIS)

In this exercise we will create a LUIS Model. As discussed in the talk, we can create our own domain specific models, but for this HOL we will use Cortana Prebuilt App. The advantage is that we can quickly see the power of Natural Language in our bot without the training. This will create a personal assistant. To see all that this prebuilt model can do see the documentation here: <https://www.luis.ai/Help/Index#PreBuiltApp>

| **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 **Cortana pre-built apps 🡪 English** 3. Once you click on your language, a Model dialog will pop up. **Copy the URL** from this screen, we will need it for our application. 4. Open up Visual Studio Code and in your project, create a file called luisai.js 5. Add the following code to the **luisai.js** file.   **------SNIP4----------------------------------------**   **var** builder **=** require('botbuilder');  *// Create bot and bind to console*  **var** connector **=** **new** builder.ConsoleConnector().listen();  **var** bot **=** **new** builder.UniversalBot(connector);  *// Create LUIS recognizer that points at our model and add it as the root '/' dialog for our Cortana Bot.*  **var** model **=** '<your models url>';  **var** recognizer **=** **new** builder.LuisRecognizer(model);  **var** dialog **=** **new** builder.IntentDialog({ recognizers: [recognizer] });  bot.dialog('/', dialog);  *// Add intent handlers*  dialog.matches('builtin.intent.alarm.set\_alarm', builder.DialogAction.send('Creating Alarm'));  dialog.matches('builtin.intent.alarm.delete\_alarm', builder.DialogAction.send('Deleting Alarm'));  dialog.onDefault(builder.DialogAction.send("I'm sorry I didn't understand. I can only create & delete alarms."));   1. Replace **‘<your models url>**’ with the **url** that we copied in step 3 2. Move over to your console and run the following  ~$ node luisai.js 3. You can ask it to create or delete alarms for you. Try different ways of saying it and notice that you can ask it the same thing in many different ways and it understands (NLP).     Right now we are just gathering intent, we are not actually performing any actions except for printing to the screen things like **‘Creating and Alarm’** or **‘Deleting an Alarm’**. Let’s add some functionality. This will allow us to inspect the intents coming back from LUIS and respond to them. We will start with the .**set\_alarm intent**  So replace the following code:   C:\Users\danie\AppData\Local\Temp\SNAGHTML75b5387.PNG  With this code:   **------SNIP5----------------------------------------------**  *// Add intent handlers*  dialog.matches('builtin.intent.alarm.set\_alarm', [  function (session, args, next) {  // Resolve and store any entities passed from LUIS.  var title = builder.EntityRecognizer.findEntity(args.entities, 'builtin.alarm.title');  var time = builder.EntityRecognizer.resolveTime(args.entities);  var alarm = session.dialogData.alarm = {  title: title ? title.entity : null,  timestamp: time ? time.getTime() : null  };    // Prompt for title  if (!alarm.title) {  builder.Prompts.text(session, 'What would you like to call your alarm?');  } else {  next();  }  },  function (session, results, next) {  var alarm = session.dialogData.alarm;  if (results.response) {  alarm.title = results.response;  }  // Prompt for time (title will be blank if the user said cancel)  if (alarm.title && !alarm.timestamp) {  builder.Prompts.time(session, 'What time would you like to set the alarm for?');  } else {  next();  }  },  function (session, results) {  var alarm = session.dialogData.alarm;  if (results.response) {  var time = builder.EntityRecognizer.resolveTime([results.response]);  alarm.timestamp = time ? time.getTime() : null;  }    // Set the alarm (if title or timestamp is blank the user said cancel)  if (alarm.title && alarm.timestamp) {  // Save address of who to notify and write to scheduler.  alarm.address = session.message.address;  alarms[alarm.title] = alarm;    // Send confirmation to user  var date = new Date(alarm.timestamp);  var isAM = date.getHours() < 12;  session.send('Creating alarm named "%s" for %d/%d/%d %d:%02d%s',  alarm.title,  date.getMonth() + 1, date.getDate(), date.getFullYear(),  isAM ? date.getHours() : date.getHours() - 12, date.getMinutes(), isAM ? 'am' : 'pm');  } else {  session.send('Ok... no problem.');  }  }]);  In the previous section of code, we are using a number of the techniques we have discussed. We are using **intent matching, text prompts, time prompts, and a waterfall**. Next we need to add an intent dialog for deleting an alarm.   Paste the following code **below** the last section in **luisai.js**.   **------SNIP6------------------------------------------------**  dialog.matches('builtin.intent.alarm.delete\_alarm', [  function (session, args, next) {  // Resolve entities passed from LUIS.  var title;  var entity = builder.EntityRecognizer.findEntity(args.entities, 'builtin.alarm.title');  if (entity) {  // Verify its in our set of alarms.  title = builder.EntityRecognizer.findBestMatch(alarms, entity.entity);  }    // Prompt for alarm name  if (!title) {  builder.Prompts.choice(session, 'Which alarm would you like to delete?', alarms);  } else {  next({ response: title });  }  },  function (session, results) {  // If response is null the user canceled the task  if (results.response) {  delete alarms[results.response.entity];  session.send("Deleted the '%s' alarm.", results.response.entity);  } else {  session.send('Ok... no problem.');  }  }  ]);  As you can see, it is very similar to the add alarm section with the addition of using the **choice prompt**. Now we need to add two more pieces to make it complete. We need to add back our default match section (one line of code) and a very simple implementation of an alarm. Paste the following code at the **bottom** of the **luisai.js** file.   **------SNIP7-----------------------------------------------**  dialog.onDefault(builder.DialogAction.send("I'm sorry I didn't understand. I can only create & delete alarms."));  // Very simple alarm scheduler  var alarms = {};  setInterval(function () {  var now = new Date().getTime();  for (var key in alarms) {  var alarm = alarms[key];  if (now >= alarm.timestamp) {  var msg = new builder.Message()  .address(alarm.address)  .text("Here's your '%s' alarm.", alarm.title);  bot.send(msg);  delete alarms[key];  }  }  }, 15000);   1. Now we can run the code. Go to your command prompt and type in the following.  ~$ node --debug luisai.js 2. Next go to Visual Studio Code and place some breakpoints in your code so you can step through and inspect the entities, as they are being set. 3. Step through the code as we had before to watch as things are running. |

## Exercise 4: Connecting to Skype and Webchat

In this exercise we are going to connect your bot to a Skype and a Webchat channel. In doing so, we ARE NOT going to be hosting our bot in the cloud but running them from our laptops. We are doing this for 3 reasons:

1. It takes out some of the complexity of the workshop by not having to troubleshoot uploading and running a node app in your cloud of choice.
2. We don’t have to walk 30 + people through setting up a cloud account.
3. It is valuable to use this technique to troubleshoot your bot prior to production so it is a valuable skill.

**SPECIAL NOTE:** An alternative for testing is to use the open source / cross platform emulator to test. You can find that here : <https://docs.botframework.com/en-us/tools/bot-framework-emulator/>   
  
The first thing we need to do is to run NGrok. You should have downloaded NGrok from <https://ngrok.com/> as it states on the website, NGrok allows you to **create “Secure tunnels to localhost”**. This means we can make the bot framework think that our bot is hosted in the cloud by having an internet accessible URL for the bot hosted on our laptop.

| **Detailed Steps** |
| --- |
| 1. Download and unzip Ngrok on your computer 2. Once unzipped double-click on the **ngrok.exe** file. This will open its own command window. 3. At the prompt in this window type the following.   ~$ ngrok http 3978  You should see the following in your command window.    We care about two things in this window. 1. The forwarding URLs (both http and https) that will be our external address for our bot to reach our local machine. 2. The Web interface address. We will use this to track traffic coming into this port for our bot.   **Leave this running**. We will need this for the rest of this walkthrough. 4. Next we need to set up our bot on the BotFramework page. Open a browser and go to <http://BotFramework.com> . You should have already set up an account at the beginning of this HOL. 5. If you are not already signed in, sign in with the MS account you created earlier. When you are signed in click on the Register a bot in the menu. 6. Fill out the bot registration form. We will go through it section by section and touch on the important fields.  **Name:** <Name of your bot> Self-explanatory **Bot Handle:** <your handle here> this will be used in the C# SDK when referencing your bot (not in Node.js SDK) **Description**: Self-explanatory 7. This next section is only two fields but many steps. The first box is the Messaging endpoint. If you were hosting you bot in the cloud, then this would be the address of the site that is hosting it. Something like http://DanielSpeakerBot.com/api.messages but since we are hosting it locally we need to use the address that Ngrok gave us when we used it.     append the address from Ngrok with /api/messages    Next you need to create an AppID and password for your bot. click on the “**Create Microsoft App ID and password**” button.     When you do this a new page will pop up and give you an **App ID**.    **SAVE THIS APP ID SOMEWHERE. WE WILL NEED IT LATER**.  Next click on the **Generate a password to continue** button.     This will pop up a modal dialog with your password.     This is the only time it will be shown.  **SAVE THIS PASSWORD ID SOMEWHERE. WE WILL NEED IT LATER** Click **ok** to continue.  Next click on the **Finish and go back to Bot Framework** button to continue. 8. In the final section, we do not need to add anything. (although in the future, setting up and using an App insights key will give you a bunch of great reporting) 9. **Agree** to the terms and click **Register** to create your bot. 10. Leave this page up and running. We will need to come back here after we modify our bot to link it to the bot framework. 11. Bring up Visual Studio Code and open up the **luisai.js** file. We will be modifying the following section of that file.     Up until now, we have been using the ConsoleConnector. Now we are going to be using the ChatConnecter. But first we need to add a node module that will help us with serving this app. It is called Restify. 12. Open up your console (Terminal, Powershell, etc..). Make sure you are in the botworkshop folder (or whatever you called it) and type the following.  ~$ npm install restify –save   **Restify** is a node module that helps make rest calls easier. 13. Once that is done, open up the luisai.js file and add the following code underneath the **var builder = require(‘botbuilder’);** code.  var restify = require('restify');  The code should look similar to this 14. Since we will be working with the new connector. Delete the connector and the bot lines so we can recreate them.    With that gone, we need to set up the restify server. To do so, add the following lines under the restify require statement.   **------SNIP8------------------------------------------------**   // Setup Restify Server  var server = restify.createServer();  server.listen(process.env.port || process.env.PORT || 3978, function () {  console.log('%s listening to %s', server.name, server.url);  });  We are setting up a server that will look for an environment variable called PORT, if it does not find one, it will start on port **3978** (which is why we used that port for NGrok).  Next we want to create the new ChatConnector (instead of the ConsoleConector).  Add the following code directly under the last code you pasted.      **------SNIP9------------------------------------------------**  // Create chat bot  var connector = new builder.ChatConnector({  appId: process.env.MICROSOFT\_APP\_ID,  appPassword: process.env.MICROSOFT\_APP\_PASSWORD  });  Notice that the connector requires and **appId** and **appPassword**. These are what we saved when we create our bots on BotFramework.com.  Now we need to new up our bot and pass in our connector like we did before and set up where the post route is (/api/messages).   Add this code below the last code you pasted.   **------SNIP10-------------------------------------------------**  var bot = new builder.UniversalBot(connector);  server.post('/api/messages', connector.listen());  There is one last step. Adding our **appId** and **appPassword**. To protect them, you should **ALWAYS** put them in **Environment Variables** (or other safe place) . If you plan to put this in production **OR** plan to save this code in github or another repository, **DO NOT** do what we are going to do right now.  Modify your connector to add your **appId** and **appPassword**.     1. Now lets run out bot. Go to your console and run the following command.   ~$ node luisai.js   You should see the restify server running on the port we specified. 2. We can test the connection by going to your bot on **BotFramework.com**   If everything works out fine when clicking the test button, you will get back an accepted message. 3. Now we can test Skype (you must have Skype installed).   In your portal, click on the Add to Skype Button    When the windows pops up, click on add you Contacts.    Once it is added to your contacts, you can chat away.     If you don’t have Skype installed you can test it out by using a webchat window. This is embedded in your bot registration page.   If you want a local web control to test with you can also add one to your project.  The first thing we need to do is to create a page to host the webchat control.   1. Open up Visual Studio Code and add a file called index.html  **------SNIP11----------------------------------------------**   <!doctype html>  <html>  <head>  <title>MyAppID</title>  </head>  <body>    </body>    </html> |
| 1. Next, go to your bot on BotFramework.com and click on the Get bot embedded codes    Click on the Web Chat Icon and then follow the link.     Click on the Add new site link    Enter the name of the site it is going on. This is just for your purposes so that you can customize per site.    This will generate both your embed code (an IFrame) and your secret keys. Click on Show on one of them so you can copy it and save it for use in the next step.     Copy the embed code into the body section on index.html you created.   Replace the YOUR\_SECRET\_KEY section with the key you saved.  Finally, to be able to access this page, you need to add a **server.get** to your **luisai.js** file.  Underneath the **server.post('/api/messages', connector.listen());** line, add the following code.   **------SNIP12-----------------------------------------------**  server.get('/', restify.serveStatic({  directory: \_\_dirname,  default: '/index.html'  }));  This will route incoming get requests (in the browser) to our **index.html** page.  To test it out, go to your console and run the following command.   ~$ node luisai.js   You should see the restify server running on the port we specified.     Next browse to <http://localhost:3978> in a browser    C:\Users\danie\AppData\Local\Temp\SNAGHTML82e3afc.PNG Now you can chat away. |
|  |

## 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.